

Programmer's Guide

Version 4.3

October 1999

CONSTRUCT SPECTRUM™ SDK

Manual Order Number: **SPV430-020IBW**

Copyright © SAGA SOFTWARE, Inc., 1999. All rights reserved.

SAGA, SAGA SOFTWARE, the SAGA logo, Free Your Information, the FYI logo, CRIS, Construct, Construct Spectrum, Construct Spectrum SDK, iXpress, Sagacertify, Sagagallery, Sagavista, and Your Fastest Route to Enterprise Integration are trademarks or registered trademarks of SAGA SOFTWARE, Inc. in the U.S. and/or other countries. Adabas, Adabas Delta Save Facility, Adabas Fastpath, Adabas SQL Server, Adabas Vista, Adaplex+, Bolero, Com-plete, Entire, Entire Access, Entire Net-work, EntireX, EntireX DCOM, Entire Broker, Entire Broker SDK, Entire Broker APPC, Entire SAF Gateway, Natural, Natural Architecture, Natural Elite, Natural New Dimension, Natural Lightstorm, Natural Vision, New Dimension, PAC, Predict, Software AG, and Super Natural are developed by Software AG of Darmstadt, Germany and are distributed in the U.S., Latin America, Canada, Israel and Japan exclusively through SAGA SOFTWARE, Inc. and its subsidiaries and distributors. Adabas and Natural are registered trademarks of Software AG of Darmstadt, Germany. Except for Adabas and Natural, these products developed by Software AG of Darmstadt, Germany are either registered trademarks or trademarks of SAGA SOFTWARE, Inc., in the U.S. and/or other countries.

Other company or product names mentioned are used for informational purposes only and may be trademarks or servicemarks of their respective owners.

TABLE OF CONTENTS

PREFACE

Prerequisite Knowledge	16
How to Use This Guide	17
If You are Creating a Construct Spectrum Web Application	17
If You are Creating a Construct Spectrum Client/Server Application	18
If You are Creating a Client/Server Application without the Client Framework. . .	19
Conventions Used in this Guide	20
Related Documentation	22
Construct Spectrum SDK	22
Construct Spectrum	23
Natural Construct	23
Year 2000 Considerations	24

1. INTRODUCTION

What is Construct Spectrum?	26
Construct Spectrum's Partner Products	26
Predict Data Dictionary and Repository	26
Middleware	26
Programming Languages	27
Development Environments	27
Construct Spectrum's Development Environments	27
The Architecture of Construct Spectrum Applications	32
Types of Applications	32
Architecture of Construct Spectrum Applications	32
Mainframe Server	34
System Functions	37
Windows	38
Internet Information Server (IIS)	40
Internet/Intranet	41

The Development Process	42
2. SETTING UP YOUR APPLICATION ENVIRONMENT ON THE MAINFRAME	
Overview	46
Setting Up Predict Definitions	47
Field Headings	47
Business Data Types	48
Default GUI and HTML Controls	48
Verification Rules	49
Default Primary Keys and Hold Fields	49
Defining a Default Primary Key	49
Defining a Default Hold Key	50
Default Business Object Description	50
Descriptive Browse Fields	51
File Volume Information in Client/Server Applications	51
Creating a Domain and Setting Up Security	52
Define a Steplib Chain	52
Define a Domain	54
Define Security for the Domain	57
3. FEATURES OF THE ABO AND WEB WIZARDS	
The Configuration Editor	60
Invoking the Configuration Editor	60
Modifying Profile Settings	62
Profile Settings	64
Copying Profiles	66
Working with Code	70
Implied User Exits	70
Preserving Customizations to Generated Code	71
Regenerating Modules	72
Regenerating Individual Modules	72
Regenerating Multiple Modules	73
Regenerating External Files	74

Editing Modules	76
Using Reports	77
Overview	77
Accessing Reports	77
Generating Reports	77
Reviewing Stored Reports	78
Using the Generate Report Dialog	79
Specifying Report Detail	81
Using Reports with a Code Comparison Tool	82
Using The Spectrum Cache	83
Overview	83
Marking Nodes to be Refreshed	84
Removing Nodes From the Spectrum Cache	85
 4. USING THE BUSINESS-OBJECT SUPER MODEL	
Overview	88
When to Use The Business-Object Super Model	89
Before Using the Business-Object Super Model	90
Check the Business-Object Super Model Defaults	90
Set up Default Values in Predict	90
Establish a Naming Convention	91
Set Up the Application Environment	92
Generating Packages Using the Business-Object Super Model	93
Step 1 — Define Standard Parameters	93
Step 2—Define General Package Parameters	95
Step 3 — Define Specific Package Parameters	98
Step 4 — Creating Another Package (Optional)	101
Step 5 — Generating the Modules	102
Generating Modules in the Construct Windows interface	102
Generating Modules in the Generation Subsystem	104

What to Do if Something Goes Wrong	105
--	-----

5. USING ACTIVEX BUSINESS OBJECTS

Overview	108
Prerequisites	109
Using the ABO Project Wizard	110
Creating the ABO Project	110
Framework Components of the ABO Project	115
Using the ABO Wizard	117
Customizing the ABO	124
Customizing the Properties Generated for the ABO	124
Using the Opt. Column.	126
Customizing With User Exits in the ABO	126
GetAppService_.SetMethodAndBlocks	127
ICSTBrowseObject_LogicalKeyInfo.Extra	127
ICSTPersist_InstanceData.Get.Extra	127
ICSTPersist_InstanceData.Let.Extra	127
ICSTPropertyInfo_PropertyInfo.Get.Extra	127
<CounterPropertyName>.Get.NullList.	128

6. USING THE SUBPROGRAM PROXY MODEL

Overview	130
System Files Containing Spectrum Administration Subsystem Data	131
Generating a Subprogram Proxy	132
Considerations for Generating With the Subprogram-Proxy Model	132
Step 1 — Specify Standard Parameters for the Subprogram Proxy	134
Step 2 — Specify the Number of Occurrences to be Returned	136
Step 3 — Add User Exits	138
Step 4 — Generate the Subprogram Proxy	139
Application Service Definition Methods	141
Accessing the Maintain Application Service Definitions Panel	141
Adding a Method	142
Step 1 — Create the Method	143
Step 2 — Update the Application Service Definition	143
Step 3 — Update the Library Image File with the Method	144

Overriding the Domain's Steplib Chain	144
Overriding Block Handling	146
Default Block Handling	146
Object Maintenance Subprogram Blocks Sent to the Server	146
Object Maintenance Subprogram Blocks Returned to the Client	147
Browse Subprogram Blocks — Sent from the Client	148
Browse Subprogram Blocks — Returned to the Client	148
Specifying Overrides	148
Step 1 — Specify Block Handling On the Server	149
Specify Block Handling On the Client	150
Versioning	151
Security Implications	151
Support for Debugging	152
 7. USING BUSINESS DATA TYPES	
Overview	154
Understanding and Using BDTs	155
Benefits of BDTs	155
Relationship With Visual Basic Data Types	155
Composition of BDTs	156
Name	156
Conversion Routine	156
Modifiers	157
Using Business Data Types	157
The BDT Controller	157
How the Client Frameworks Use BDTs	158
Calling Conversion Routines	159
Convert to Display	160
Convert from Display	160
Converting In-Place	161
Creating Sample Values	162
Using Modifiers	162
BDTs and Natural Formats	163
Handling Errors Returned from a BDT Conversion Routine	164
How Spectrum Web Applications Use BDTs	165
BDTs Supplied With Construct Spectrum	166

Alpha	167
Boolean	167
Time	168
Numeric	169
Currency	170
Date	170
Referencing BDTs in Predict	171
Defining BDTs	172
Name	172
Modifiers	172
Supported Natural Formats	173
Returning An Appropriate Variant Type	173
Returning Conversion Error Information	174
Runtime Error Handling	174
Creating and Customizing BDTs	176
BDTs and the Client/Server Framework	176
Understanding the BDT Objects	176
Creating BDT Conversion Routines	178
Registering Your BDT	181
Creating a Natural-to-BDT Mapper	182
Other Considerations	184
One Conversion Routine — Multiple BDTs	184
Where To Place The Conversion Routine	184
BDT Overrides	185
Referencing BDTs in Your Application	186
BDTs and the Web Framework	187
Understanding the BDT Objects	187
Implementing BDTs in the Web Framework	189
What is a BDT Class	189
Registering BDTs	189
Use the Windows Registry to Register BDT Classes	190
Explicitly Register BDT Classes	191
Conversion Routine	192
Creating the BDT Class	195
Other BDT Controller Methods	196
Creating a Natural-to-BDT Mapper	197

Other Consideration	199
One BDT Class — Multiple BDTs	200
8. DEBUGGING YOUR CLIENT/SERVER APPLICATION	
Overview	202
Communication Errors	202
Communication Error Handling	203
Traditional Debugging Tools	204
Construct Spectrum Debugging Tools	204
Types of Errors	206
Visual Basic Runtime Errors	206
Communication Errors	207
Natural Runtime Errors	207
Construct Spectrum-Related Errors	207
Errors that Do Not Return an Error Message	207
Generating Debug Data	208
Saving Parameter and Debug Data	208
Setting Trace Options	208
Trace-Option(1)	209
Creating Debug Data	211
Trace-Option(2)	213
Specifying Where Debug Data Should Be Saved	213
Accessing the Maintain User Table Panel	214
Running Spectrum Dispatch Services Online	217
Using the INPUT Statement as a Debugging Tool	217
Simulating Client Calls to Use Natural Debugging Tools	219
Invoking Subprogram Proxies Online	219
Accessing the Invoke Proxy Function	220
Debugging Tools on the Client and Server	223
Diagnostics Program	223
The Translations Program	227
Troubleshooting Quick Reference	229
Registry Usage	229
SDC.ini	230
SDCApp.ini	230
Checking for Necessary DLLs	231

Construct Spectrum Add-In	231
Useful SDC Properties	231
Application Object	232
NaturalDataArea Object	232
Dispatcher Object	233
RequestProperty Properties	233

9. DEPLOYING YOUR CLIENT/SERVER APPLICATION

Data Transfer	238
Data Transfer Utilities	238
Manual Data Transfer	238
Distributing Your Application	239
Step 1 — Creating the Executable	239
Step 2 — Collecting Files For Installation	239
Step 3 — Installing the Client Application	240
Step 4 — Running the Application	240

10. UNDERSTANDING THE SPECTRUM DISPATCH CLIENT

Overview	244
Process of Creating Applications	245
Step 1 — Create Parameter Data Area Instances	245
Step 2 — Assign Values to the Fields in the Parameter Data Areas	246
Step 3 — Use the CallNat Method on the Client	246
Step 4 — Check the Success of the CALLNAT	247
Summary	248
Spectrum Dispatch Client Components	249
Natural Data Area Simulation	250
Data Area Definitions	250
Data Area Simulation Objects	252
The Application Object	254
Creating NaturalDataArea Objects	255
The NaturalDataArea Class	255
What Else Should You Know About the NaturalDataArea Object?	259
Field Names are Not Case-Sensitive	259
Alpha Fields	260
Fully-Qualified Field Names	260

Redefined Fields	260
Checking Natural Fields During Compile.	261
Reading Arrays and Structures	262
Runtime Errors.	263
The DataDefinitionArea Class.	263
The NaturalFieldDef Class	264
Client/Server Communication	268
Level 1 Block Optimization	269
Application Service Definitions	271
Dispatcher Objects and Dispatch Service Definitions	274
Service Selection.	276
Remote Subprogram Invocation.	277
Timeout, Retry, and Resume Handling.	278
Compression and Encryption.	282
Tracing	282
Database Transaction Control.	283
Error Reporting.	284
User Identification and Authentication	285
Library Image Files and the Steplib Chain	286
Advanced Features.	287
The FieldRef Property	287
1:V Fields	293

11. CREATING SPECTRUM APPLICATIONS WITHOUT THE CLIENT FRAMEWORK

Setting Up the Server Components.	298
Creating or Selecting Application Services.	298
No Terminal I/O	298
Subprogram Interface	299
No Global Data Area (GDA).	299
Parameter Data Area (PDA) Data Size Limitation	299
Subprogram Behavior	299
Externalize Parameters	299
Timing Issues	300
Example of Creating a Simple Natural Subprogram	300

Generating Subprogram Proxies Using the Spectrum Models	303
Using the Model	303
Application Service Definition	305
Creating the Library Image Files (LIFs)	307
Using the Construct Spectrum Add-In	307
Before You Start	307
Downloading Definitions	308
Developing the Client Application	311
Step 1 — Create a New Project	312
Step 2 — Add a Reference to the Object Library	312
Step 3 — Write Code to Initialize the Spectrum Dispatch Client	313
Step 4 — Create the User Interface	316
Step 5 — Write the Code that Does the Call	317
Step 6 — Run the Application	318
APPENDIX A: GLOSSARY.....	321
APPENDIX B: UTILITIES.....	341
SPUREPLY Subprogram	342
Features and Benefits	342
Limitation	342
Supported Methods	342
Message Protocol	343
Call Interface	344
Data Areas	345
SPAREPLY Data Area	345
SPAREPM Data Area	347
Call Example	348
Send Buffer Example	348
SPUETB – Spectrum Interface Subprogram	350
Features and Benefits	350
Broker Error Handling	350
Error Logging	350
Shutdown Requests	351
Server Timeouts	351
Command Handling	351

SPUETB Interface	351
Data Areas	352
SPAETB Data Area	352
ETBCB Data Area	358
SEND-BUFFER	359
RECEIVE-BUFFER	359
RESERVED-AREA	359
CDPDA-M	359
Using SPUETB	359
CMD TRACE	359
Enabling Tracing	360
Example	361
Disabling Tracing	361
Trace Response	361
Testing Trace Facility	361
CMD CALLNAT	361
The Conversation Factory	363
SPUTLATE — Character Translation	364
Determining a Character Set — SPUASCII	364
Multi-Tasking Verification Utility	365
Log Utilities	366
Spectrum Log Utilities	366
Construct Spectrum Control Record Log Utilities	367
Domain Log Utilities	368
Spectrum Group Log Utilities	369
Application Service Definition Log Utilities	370
Spectrum Steplib Log Utilities	371
User and Group Log Utilities	372
INDEX	373

PREFACE

Welcome to the Construct Spectrum software development kit (SDK), which uses Visual Basic on the client and Natural Construct on the server. The *Construct Spectrum Programmer's Guide* helps developers create client/server and web applications and customize, debug, and deploy applications.

This preface will help you get the most out of the guide and find other sources of information about creating Construct Spectrum applications.

The following topics are covered:

- **Prerequisite Knowledge**, page 16
- **How to Use This Guide**, page 17
- **Conventions Used in this Guide**, page 20
- **Related Documentation**, page 22
- **Year 2000 Considerations**, page 24



Prerequisite Knowledge

The *Construct Spectrum Programmer's Guide* does not provide information about the following topics. We assume that you are either familiar with these topics or have access to other sources of information about them.

- Natural Construct
- Microsoft® Visual Basic®
- Predict®
- Natural® programming language and environment
- Entire Broker™
- Entire Net-Work®

How to Use This Guide

The *Construct Spectrum Programmer's Guide* provides information about core development tasks that the majority of Construct Spectrum users perform, whether they are:

- Creating Construct Spectrum web applications
- Creating Construct Spectrum client/server applications
- Creating client/server applications without using Construct Spectrum's client frameworks

The following sections explain how to use this and related guides to perform these types of tasks.

If You are Creating a Construct Spectrum Web Application

If you wish to use Construct Spectrum's tools to create all the components of a web application, we recommend that you read:

- Chapter 1, **Introduction** for an overview of the product, development process, and the applications you can develop.
- Chapter 2, **Setting Up Your Application Environment on the Mainframe** for detailed information about how to define domains and security options to control what data users of your application will access on the mainframe.
- Chapter 3, **Features of the ABO and Web Wizards** for information about setting configuration options for the wizards, using the client-side cache, and modifying code frames.
- Chapter 4, **Using the Business-Object Super Model** for detailed information about how to use this model wizard to generate the Natural components of your application.
- Chapter 5, **Using ActiveX® Business Objects** for detailed information about creating ABOs and an ABO project to contain them using the wizards supplied with Construct Spectrum.
- *Developing Web Applications* for detailed information about creating the web components of your application.

As you customize and regenerate your application components, you will find these chapters in the *Construct Spectrum Programmer's Guide* useful:

- Chapter 6, **Using the Subprogram Proxy Model**
- Chapter 7, **Using Business Data Types**

If You are Creating a Construct Spectrum Client/Server Application

If you wish to use Construct Spectrum's tools to create a client/server application to run on Windows® 95, Windows 98, Windows 2000 or Windows NT, we recommend that you read:

- Chapter 1, **Introduction** for an overview of the product, development process, and the applications you can develop.
- Chapter 2, **Setting Up Your Application Environment on the Mainframe** for detailed information about how to define domains and security options to control what data users of your application will access on the mainframe.
- *Developing Client/Server Applications* for detailed information about using the VB-Client-Server super model to generate all of your application's components. It explains how to set up a Visual Basic project and customize maintenance and browse dialogs. Turn to this guide, as well, if you wish to move existing server-based applications to the Construct Spectrum client/server architecture.

As you customize and regenerate applications components, you will find these chapters in the *Construct Spectrum Programmer's Guide* useful:

- Chapter 6, **Using the Subprogram Proxy Model**
- Chapter 7, **Using Business Data Types**
- Chapter 8, **Debugging Your Client/Server Application**
- Chapter 9, **Deploying Your Client/Server Application**

If You are Creating a Client/Server Application without the Client Framework

If you wish to create a client/server application without using Construct Spectrum's client framework, we recommend that you read:

- Chapter 1, **Introduction** for an overview of the product, development process, and applications you can develop.
- Chapter 10, **Understanding the Spectrum Dispatch Client** for detailed information about the role of the SDC in client/server communication.
- Chapter 11, **Creating Spectrum Applications Without the Client Framework** for step-by-step procedures for creating your application.



Conventions Used in this Guide

This guide uses the following typographical conventions:

Example	Description
Introduction	Bold text in cross references indicates chapter and section titles.
“A”	Quotation marks indicate values you must enter.
Browse model, GotFocus, Enter	Mixed case text indicates: <ul style="list-style-type: none">• The names of Natural Construct and Construct Spectrum editors, fields, files, functions, models, panels, parameters, subsystems, variables, and dialogs.• The names of Visual Basic classes, constants, controls, dialogs, events, files, menus, methods, properties, and variables.• The names of keys.
Alt+F1	A plus sign (+) between two key names indicates that you must press the keys together to invoke a function. For example, Ctrl+S means hold down the Ctrl key while pressing the S key.
CHANGE-HISTORY	Uppercase text indicates the names of Natural command keywords, command operands, data areas, help routines, libraries, members, parameters, programs, statements, subprograms, subroutines, user exits, and utilities.
<i>Construct Spectrum Administrator's Guide, variable name</i>	Italicized text indicates: <ul style="list-style-type: none">• Book titles.• Placeholders for information you must supply.

Example	Description (continued)
<code>[variable]</code>	In syntax and code examples, values within square brackets indicate optional items.
<code>{WHILE UNTIL}</code>	In syntax examples, values within brace brackets indicate a choice between two or more items; each item is separated by a vertical bar ().



Related Documentation

The documentation sets for Construct Spectrum and Natural Construct consist of the following manuals:

Construct Spectrum SDK

- *Construct Spectrum Programmer's Guide*
This guide is for developers creating Natural modules and ActiveX Business Objects to support applications that will run in the Natural mainframe environment and a Windows environment and/or an internet server.
- *Developing Web Applications*
This guide is for developers creating the web components of applications. It explains how to use the Construct Spectrum wizards in Visual Basic to generate HTML templates, page handlers, and object factory entries. It also contains detailed information about customizing, debugging, deploying, and securing web applications.
- *Construct Spectrum Reference Manual*
This manual is for application developers and administrators who need quick access to information about Construct Spectrum application programming interfaces (APIs) and utilities.
- *Construct Spectrum Messages*
This manual is for application developers, application administrators, and system administrators who wish to investigate messages returned by Construct Spectrum run-time and SDK components.
- *Developing Client/Server Applications*
This guide is for developers creating client components for applications that will run in the Natural mainframe environment (server) and a Windows environment (client).

Construct Spectrum

- *Construct Spectrum and SDK Client Installation*
This manual explains how to install and set up the Construct Spectrum run-time and SDK components on the client.
- *Construct Spectrum and SDK Mainframe Installation*
This manual explains how to install and set up the Construct Spectrum run-time and SDK components on the mainframe.
- *Construct Spectrum Administrator's Guide*
This guide is for administrators who wish use the Construct Spectrum Administration subsystem to set up and manage Construct Spectrum applications.

Natural Construct

- *Natural Construct Installation and Operations Manual for Mainframes*
This manual provides essential information for setting up the latest version of Natural Construct, which is needed to operate the Construct Spectrum programming environment.
- *Natural Construct Generation User's Manual*
This manual explains how to use the Natural Construct models to generate applications that will run in a mainframe environment.
- *Natural Construct Administration and Modeling User's Manual*
This manual explains how to use the Administration subsystem of Natural Construct and how to create new models.
- *Natural Construct Help Text User's Manual*
This manual explains how to create online help for applications that run on server platforms.



Year 2000 Considerations

SAGA SOFTWARE strongly recommends that you work with the preferred Y2K-capable date format of alphanumeric eight (A8) and the numeric eight (N8) format. Although alphanumeric six (A6) date formats and numeric six (N6) date formats can still be used for some functionality, full functionality, inclusive of Y2K capability, is contingent upon the use of the A8 and N8 date formats. A8, A6, N8 and N6 formats are used in examples throughout the documentation.

Year 2000 capability as it relates to Natural Construct, Construct Spectrum, and Construct Spectrum SDK requires the use of the A8 and N8 date formats.

With regard to products or services, SAGA SOFTWARE defines Year 2000 “readiness” or “capability”, or the fact that a product or service is Year 2000 “ready” or “capable”, to mean that the product or service in question is capable of accurately processing, providing and receiving data from, into and between the twentieth and twenty-first centuries, and that it will correctly create, store, process and output information related to or including dates on or after January 1, 2000, provided that all other products, including hardware or software, used in combination with the product or service, properly exchange date information with the product or service.

INTRODUCTION

This chapter explains the components of Construct Spectrum and the architecture of the applications you can create with the software development kit (SDK). An overview of the general steps involved in developing applications will prepare you for the detailed procedures in this and related guides.

The following topics are covered:

- **What is Construct Spectrum?**, page 26
- **The Architecture of Construct Spectrum Applications**, page 32
- **The Development Process**, page 42

What is Construct Spectrum?

Construct Spectrum and the software development kit (SDK) comprise a set of middleware and framework components, as well as integrated tools that use the specifications you supply to generate all the components of distributed applications.

Construct Spectrum comprises two products:

- The software development kit (SDK) is a set of tools, wizards, and framework components with which you can build client/server and web applications.
- Construct Spectrum is a middleware product that facilitates communication between client and server.

Construct Spectrum's Partner Products

Construct Spectrum works with other products, some of which you may not be aware of as you build applications. You may wish to explore them as you become experienced at building Construct Spectrum applications.

Predict Data Dictionary and Repository

Construct Spectrum works closely with Predict, a data dictionary and repository that manages metadata about the information contained in the database that your applications use. Predict's "views" of data and the relationships between data help you define the business objects your applications access and maintain. Predict's verification rules and keywords are used to validate and format data, and its field definitions automatically select controls for your applications. You can do a lot with Predict to define the defaults Construct Spectrum uses to generate your applications.

Middleware

Construct Spectrum uses Entire Broker, either with Entire Net-Work or configured to use TCP/IP, to communicate between the client and server components of the application.

Your applications also use Construct Spectrum's middleware components — the Spectrum Dispatch Client (SDC) and Spectrum dispatch service — to encapsulate calls to Entire Broker on the client and server and to perform such functions as data translation, encryption, and compression. When the client makes a communication request, the SDC translates the request into a compact, secure message and transmits it to the server via Entire Broker. On the server, the Spectrum dispatch service converts the incoming request for processing by the server application while enforcing multi-level security. Construct Spectrum then uses a similar technique to return the processed result to the client.

Programming Languages

Construct Spectrum applications incorporate Natural and Visual Basic code. You can also develop client/server applications using other OLE-compliant languages. If you are creating web applications, your web pages will use JavaScript™ and HTML, including Construct Spectrum's specialized replacement tags for presenting data dynamically in web pages.

Development Environments

Besides Construct Spectrum's development environments, the product provides tools that are integrated with the Natural and Visual Basic development environments so that you can take advantage of the functionality of each. For example, you may wish to use Natural's powerful code editors or Visual Basic's sophisticated debugging facilities. For more information, see the following section.

Construct Spectrum's Development Environments

As you develop your applications, you will work in at least three environments: the Construct Spectrum Administration subsystem, the Construct Windows interface, and Visual Basic, using the Construct Spectrum Add-Ins.

In the Construct Spectrum Administration subsystem on the mainframe, you can manage system and application data for your applications.

```

BS_MAIN   ***** Construct Spectrum Administration Subsystem *****   CDLAYMN1
Jul 30                                - Main Menu -                                10:14 AM

                                Functions
                                -----
                                SA   System Administration
                                AA   Application Administration

                                ?   Help
                                .   Terminate
                                -----

Function ..... _

Command .....
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---
      help  retrn quit          flip                                main

```

Construct Spectrum Administration Main Menu

In the Construct Windows interface running on your PC, you can use wizards to generate Natural and Visual Basic modules for your application:

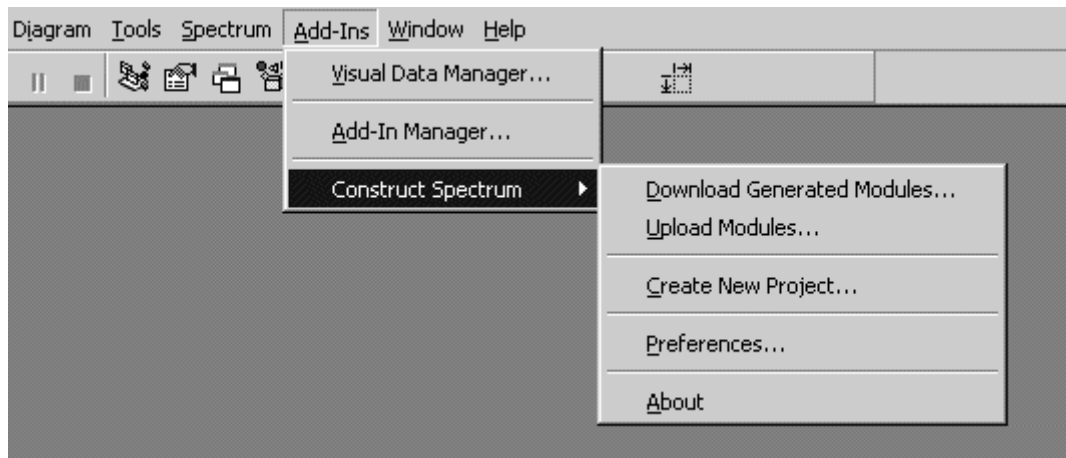


New Specification Window — Construct Windows Interface

The wizards available in the Construct Windows interface are also available in the Generation subsystem in your Natural Construct mainframe environment.

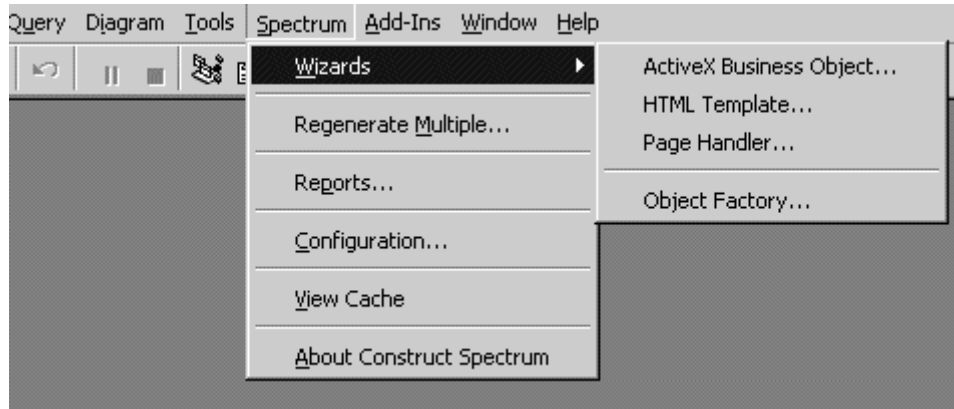
In Visual Basic, you will use the Construct Spectrum Add-Ins to create projects, work with Visual Basic modules, and generate ActiveX business objects and web components.

If you are creating a client/server application, you will use the Construct Spectrum options on the Add-Ins menu to create projects, download modules from the main-frame server, and set configuration options:



Construct Spectrum Options — Add-Ins Menu

If you are creating a web application, you will use the Construct Spectrum ABO project and the web project wizards to create projects. You will also use the Spectrum menu to invoke wizards that generate web components, regenerate modules, view the cache of server data, etc.



Spectrum Menu in Visual Basic

If you are creating a web application, you will also be working with an HTML editor of your choice, the Microsoft® Management Console® to manage your Microsoft Internet Information Server® on Windows NT, and/or the Personal Web Server® if you are using Windows to develop applications.

Information about how to access and use these environments is presented where you need it throughout the documentation.

The Architecture of Construct Spectrum Applications

The following sections explain the types of applications you can create with Construct Spectrum and their components.

Types of Applications

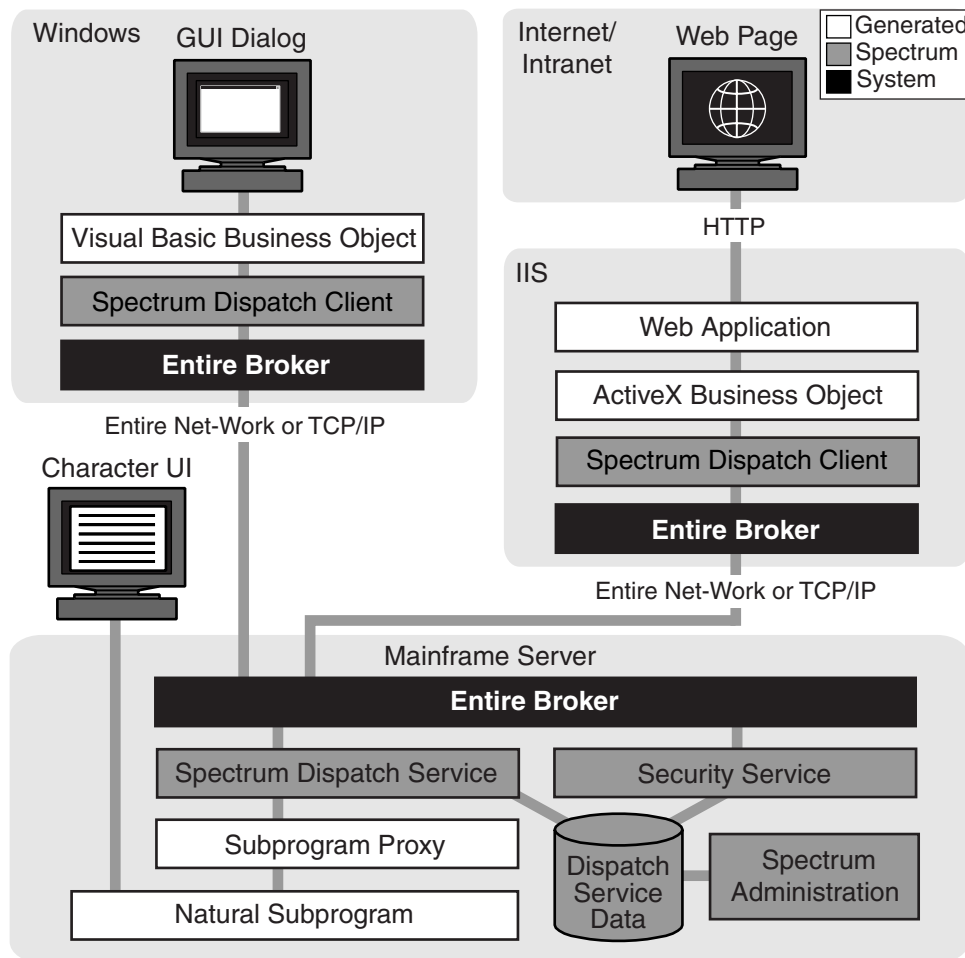
Using Construct Spectrum SDK, you can create two kinds of applications:

- Client/server applications that run on Windows or Windows NT (client) and access Natural components and data on a mainframe (server).
Client/server applications are composed of Natural modules that encapsulate maintenance and query functions on the server, Visual Basic components that function on the client and present the user interface, and run-time components that communicate between client and server.
- Web applications that run on IIS and can be accessed with Microsoft Internet Explorer® and Netscape Navigator®.
Web applications are composed of Natural modules that encapsulate maintenance and query functions on the server, ActiveX business objects that communicate between client and server components, page handlers that manage the processing of HTML templates, and HTML templates that present web pages.

This guide explains how to develop components and functionality that are common to the different types of applications. Information specific to client/server applications is contained in *Developing Client / Server Applications*. Information specific to web applications created with Construct Spectrum is contained in *Developing Web Applications*.

Architecture of Construct Spectrum Applications

The following diagram shows the architecture of Natural character-based applications, client/server applications, and web applications.



Architecture of Construct Spectrum Applications

The following sections explain these components according to the platforms on which the components run: mainframe server, Windows, IIS, and internet or intranet.

Mainframe Server

Component	Description
Natural subprogram	<p>Natural subprograms perform maintenance and browse functions on the mainframe server. The same set of business objects can be accessed from character-based Natural applications, client/server applications, and web applications. This ensures that the integrity of business data is preserved, independent of the presentation layer.</p> <p>Natural subprograms may be either generated by Construct models or written by hand. The models that generate subprograms and their parameter data areas (PDAs) are the VB-Client-Server-Super-Model, Business-Object-Super-Model, Object-Maint-Subprogram model, and Object-Browse-Subprogram model.</p> <p>Natural subprograms can also be written by hand, provided they follow certain guidelines. For example, screen I/O functions are not allowed, and records cannot be held between conversations.</p>
Character UI	Non-distributed Natural applications created with Natural Construct accessing subprograms directly.
Subprogram proxy	<p>A subprogram proxy acts as a bridge between a specific subprogram and the Spectrum dispatch service. The subprogram proxy:</p> <ul style="list-style-type: none"> • provides a common interface so that the Spectrum dispatch service can pass the same set of parameters to any subprogram proxy • issues a CALLNAT to the subprogram • converts the parameter data of the subprogram into a format that can be transmitted between the client and server

Component	Description (continued)
Spectrum dispatch service	<ul style="list-style-type: none"> • supports optimization of the data passed through the network so that only input parameters need to be sent to the Spectrum dispatch service and only output parameters need to be returned to the client • validates the format and length of the data received from the client • supports debugging features to help uncover inconsistencies between the data sent by the client and the data expected by the subprogram proxy <p>For more information, see Using the Subprogram Proxy Model, page 129.</p>
	<p>The Spectrum dispatch service provides a common interface and Entire Broker services for Natural subprograms in the application. The main functions of the Spectrum dispatch service are to:</p> <ul style="list-style-type: none"> • receive requests from the client by way of Entire Broker • optionally decompress or decrypt (or both) and translate the request message (see System Functions, page 37) from the client's character set (ASCII) to the server's character set (either ASCII or EBCDIC) • check Security to ensure that the client is allowed to issue such a request • determine the name of the subprogram proxy that handles the request • issue a CALLNAT to the subprogram proxy, passing the received message as a parameter string • optionally compress, encrypt (or both) the message to be returned (see System Functions, page 37) • send information received from the subprogram proxy back to the client application

Component	Description (continued)
Dispatch service data	The information defined and maintained in the Spectrum Administration subsystem is accessed by Spectrum dispatch services anywhere on the network by way of Entire Broker.
Spectrum administration	This mainframe subsystem allows system administrators, application administrators, and application developers to set up and manage system and application environments. For more information, see <i>Construct Spectrum Administrator's Guide</i> .
Security service	A Spectrum security service checks client requests against the security settings defined in the Construct Spectrum Administration subsystem. This stand-alone service operates independently of any one Spectrum dispatch service. Its independence allows the security service to process, in one central location, the requests of several Spectrum dispatch services, which may be located on nodes throughout the network. For more information about security services and security settings, see <i>Construct Spectrum Administrator's Guide</i> .
Entire Broker	Entire Broker transfers messages between Windows or the web server and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.

System Functions

All Spectrum dispatch services defined in the Construct Spectrum Administration subsystem have access to common system functions:

Function	Description
Return debugging information	Ensures that all requested debugging information is generated into the source area. Debugging information is requested by setting a Trace-Option in the subprogram proxy. The debugging information is stored as a source member that can be examined or used to initiate the request locally on the server, removing the client and the network from the test. See Debugging Your Client/Server Application , page 201 for more information about trace options.
Encrypt and decrypt data	Supplies an interface that can be called by the external (assembler or C) routines used to encrypt and decrypt data.
Compress and decompress data	Supplies an interface that can be called by the external (assembler or C) routines used to compress and decompress data.
Error handling	Manages the capturing of runtime errors, returning the errors to the client. If possible, this function also restarts the service that ended with the runtime error.
Message handling	Returns a message string based on a message number and substitution values. The function accepts and updates the data used by the Spectrum dispatch service to return the message.
Data translation	Translates data received from the client into EBCDIC or ASCII, depending on the requirements of the server.

Windows

Construct Spectrum client/server applications run on Windows or Windows NT.

Component	Description
Entire Broker	Entire Broker transfers messages between the client and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.
Spectrum Dispatch Client (SDC)	This Component Object Model (COM) middleware component enables Construct Spectrum applications to read from, and write to, variables in a Natural parameter data area (PDA) and to issue CALLNAT statements to Natural subprograms.

The main functions of the Spectrum Dispatch Client are:

- Natural parameter data area simulation
The Spectrum Dispatch Client defines the parameter data of Natural business objects as a series of Natural data fields, which may include structures, arrays, and redefines. To call a business object, the Construct Spectrum application must be able to assign values to these parameter data fields before calling the business object and then read the fields after the data is returned from the server.

To facilitate this, the Spectrum Dispatch Client simulates Natural parameter data areas, allowing the application developer to create code that allocates a data area and reads and writes the fields in the data area. Natural parameter data areas residing in a library on the server may be downloaded (by the Construct Spectrum Add-In) to the client. This lets the Spectrum Dispatch Client know the structure (field names and formats) of a parameter data area. Parameter data areas are stored in the library image file on the client and only need to be downloaded after creation or whenever they change on the server.

Component	Description (continued)
	<ul style="list-style-type: none"> • CALLNAT simulation The Spectrum Dispatch Client allows a Construct Spectrum application to issue a CALLNAT to a Natural subprogram. All that needs to be specified in the client code is the logical name of the subprogram to be called and the list of parameter data areas to pass to the subprogram. • Encapsulation of Entire Broker calls The Spectrum Dispatch Client uses Entire Broker calls to communicate with the Spectrum dispatch service. These calls are not exposed within the application layer, so the developer never needs to code Entire Broker calls. • Database transaction control Often, two or more calls to subprograms occur within the same database transaction such that an END TRANSACTION statement can be issued if all calls complete successfully. Also, it is advantageous to have the client application control the point at which the END TRANSACTION or BACKOUT TRANSACTION statement occurs. The Spectrum Dispatch Client and the Spectrum dispatch service cooperate to provide these capabilities.
	<p>For more information, see Understanding The Spectrum Dispatch Client, page 243.</p>
Visual Basic business object	<p>A Visual Basic business object is a Visual Basic class that acts as an intermediary between a dialog and the Spectrum Dispatch Client. This class invokes the methods of subprograms on behalf of dialogs and instantiates all the data areas required to communicate with a subprogram. Visual Basic business objects can also perform local data validation to provide immediate feedback to the user without involving a network call.</p>
GUI dialog	<p>GUI dialogs represent graphical interface screens that communicate with the user and interact with the Visual Basic business objects and other framework components to implement business processes.</p>

Internet Information Server (IIS)

Web applications created with Construct Spectrum work with IIS.

Component	Description
Entire Broker	Entire Broker transfers messages between the web server and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.
Spectrum Dispatch Client	This Component Object Model (COM) middleware component enables web applications to read from, and write to, variables in a Natural parameter data area (PDA) and to issue CALLNAT statements to Natural subprograms. Its main functions are simulating PDAs and CALLNATs, encapsulating Entire Broker calls, and controlling database transactions. As the client counterpart of Spectrum dispatch services, it is also responsible for such things as data marshaling, encryption, compression, error-handling, and all Entire Broker communication.

For more information, see **Understanding The Spectrum Dispatch Client**, page 243.

Component	Description (continued)
ActiveX Business Object	<p>Each back-end business object is represented on the web server as an ActiveX object. This object encapsulates all of the communication with the Spectrum Dispatch Client, making it efficient to invoke Natural services from the client.</p> <p>For more information, see Using ActiveX Business Objects, page 107.</p>
Web application	<p>A Construct Spectrum web application consists of framework components supplied with all Construct Spectrum web projects and components that you generate using Construct Spectrum wizards. Generated components are HTML templates, page handlers, and object factory entries.</p> <p>For more information, see Architecture of a Construct Spectrum Web Application, page 24, <i>Developing Web Applications</i>.</p>

Internet/Intranet

Construct Spectrum web applications support Internet Explorer and Netscape Navigator browsers at version 4.0 and later.

The Development Process

This section explains, in general terms, the steps involved in developing a Construct Spectrum application. For detailed information about developing a particular type of application, see the following sources:

- For an overview of developing web applications, see **The Development Process**, page 31, *Developing Web Applications*.
- For an overview of developing client/server applications, see **The Development Process**, page 33, *Developing Client/Server Applications*.
- For an overview of developing client/server applications without the Construct Spectrum client framework, see **Creating Spectrum Applications Without the Client Framework**, page 297.

The steps involved in developing a Construct Spectrum application are:

- 1 Plan your application
You will save time and effort by planning as completely as possible the purpose, functionality, security, and user interface of your application.
- 2 Set up your application environment
Based on the functionality you planned for your application, ensure that the file, field, and relationship definitions in Predict are set up to support the business objects and business rules your application will use. This step also takes you to the Construct Spectrum Administration subsystem, where you will set up a domain and steplib chain so that the application can access the appropriate data. You may also wish to define users, groups, and security settings in this step.
- 3 Generate application components
Using Construct Spectrum models and/or wizards, enter the specifications for your application components and generate them. For the first iteration of your application, we recommend using the super model wizards to create multiple components. For Natural modules and client/server Visual Basic components, you can use either the models in the Natural Construct Generation subsystem on the mainframe or the model wizards in the Construct Windows interface. If you are creating a web application, you will also use the wizards Construct Spectrum adds to Visual Basic. This step also involves creating new Visual Basic projects and populating them with components.
- 4 Customize, test, and debug the application
Once you have created the basis of your application, you can customize its look and functionality. This iterative process could require you to regenerate modules using the individual models Construct Spectrum supplies.

5 Deploy the application

When your application is fully functional, you are ready to distribute it to users. This step can involve creating an installation kit and deploying the Construct Spectrum Administration subsystem.

SETTING UP YOUR APPLICATION ENVIRONMENT ON THE MAINFRAME

This chapter describes the tasks you must perform on the mainframe before generating a client/server or web application.

Before performing any of the tasks described in this section, ensure that all required software has been installed and configured on your server and client. For information about installing the required software, see *Construct Spectrum and SDK Mainframe Installation* and *Construct Spectrum and SDK Client Installation*.

The following topics are covered:

- **Overview**, page 46
- **Setting Up Predict Definitions**, page 47
- **Creating a Domain and Setting Up Security**, page 52

Overview

Before you can generate applications, you must complete some set-up tasks to ensure that your application accesses the database records it needs and that users will be able to access the application. The following tasks are involved:

- Set up file and field definitions in Predict. You can also affect how field names and controls are derived and how validations are performed by adjusting Predict settings.
- Create and associate a steplib chain and domain in the Construct Spectrum Administration Subsystem.
- Set up security privileges for the domain. This involves defining users and groups and linking them to the domain in the Construct Spectrum Administration subsystem.

This chapter explains these steps in more detail.

Setting Up Predict Definitions

With any application created with Natural Construct or Construct Spectrum, you must set up file and field definitions in Predict. This includes setting up your application files and defining their intra- and inter-object relationships.

For information about these tasks, see **Design Methodology**, page 231, and **Use of Predict in Natural Construct**, page 941, of the *Natural Construct Generation User's Manual*.

The Predict features that have special implications for Construct Spectrum applications include:

- Field headings
- Business Data Types
- Default GUI and HTML controls
- Verification rules
- Primary keys and hold fields
- Default business object description
- Descriptive browse fields

Tip: You can postpone the set-up tasks described in this section until a later iteration of your application. These tasks may be optional and, in all cases, Construct Spectrum applies its own values for these set-up items based on your existing Predict file and field definitions.

Field Headings

If a field definition has a heading in Predict, the heading is used to derive the caption for the control on the dialog or page. If no heading is coded in Predict, the caption is generated by converting the field name to mixed case and changing special characters (dashes and underscores) to spaces.

When creating a client/server application, you can change the captions on the form in Visual Basic.

When creating a web application, you can modify captions in the HTML Template wizard in Visual Basic. For more information, see **Generating and Customizing HTML Templates**, page 105, *Developing Web Applications*.

Business Data Types

Business data types (BDTs) associate additional formatting with data fields to help ensure that data is presented consistently and validated in your application. By default, generated modules implement basic format and length-checking to ensure that all values stored on the client are of legal format and length. Business data types extend this concept by allowing the use of user-defined data types related to business representations of the data. For example, a numeric field might be intended to store a currency amount, a net weight, a date, or a quantity. Each of these values might be presented to the user and validated in a different way, although they are all defined as numeric fields. For example, a credit card number could be stored on the database as a 16-digit value. However, when this value is placed on a page, it could be shown using the format 9999-9999-9999-9999. Furthermore, the user could update the value with or without the dashes, and the BDT will ensure that the unformatted value is assigned back to the database.

To associate a database field with a business data type, assign a special BDT keyword to the field in Predict. For more information, see **Using Business Data Types**, page 153.

Default GUI and HTML Controls

Construct Spectrum applies complex derivation rules to determine the most appropriate control to represent a database field. Nevertheless, there may be times when the default control is not ideal for a particular application. In these cases, you can override the default control by assigning the database field a special keyword. If you are creating a web application, you can change some controls in the HTML Template wizard.

For more information, see **Overriding GUI Controls**, page 160, *Developing Client/Server Applications* or **Generating and Customizing HTML Templates**, page 105, *Developing Web Applications*.

Verification Rules

Verification rules are used to force the application user to make a selection based on one or more predetermined choices. For example, if your application has a field where a valid month must be entered, you can specify a verification rule for the field so that only a valid month will be accepted.

One criteria that Construct Spectrum uses to determine the most appropriate GUI or HTML control for a particular field is the presence of verification rules attached to the field. In the previous example of presenting valid months, Construct Spectrum would attach a drop-down combo box to the field in the dialog or page. The user could select a valid value from the drop-down combo box.

For more information, see **Overriding GUI Controls**, page 160, *Developing Client/Server Applications* or **Generating and Customizing HTML Templates**, page 105, *Developing Web Applications*.

Default Primary Keys and Hold Fields

Predict keywords can also be used to designate default primary key values and logical hold fields when using a super model. This reduces the number of specifications you must enter when using the super model.

Defining a Default Primary Key

To define a default primary key, specify a descriptor name in the Sequence field for the file in Predict. Natural Construct observes the following priorities when defaulting a primary key name for a file:

- 1 If the value of the default Sequence field for the file is unique and a valid descriptor, Natural Construct uses this value as the primary key.
- 2 If the value of the default Sequence field is not unique, Natural Construct reads through the file and uses a unique descriptor field value as the primary key.
- 3 If the file does not have a unique descriptor field, but has only one descriptor field, Natural Construct assumes this field value is unique and uses it as the primary key.

Defining a Default Hold Key

To define a default logical hold field, attach a keyword called “HOLD-FIELD” to the field in Predict.

Note: You may have to first define the HOLD-FIELD keyword in Predict using Keyword Maintenance.

Natural Construct observes the following priorities when defaulting a hold field name for a file:

- 1 If the HOLD-FIELD keyword is attached to a field that meets the format criteria for a hold field, Natural Construct uses this field as the logical hold field.
- 2 If a field name contains any of the following strings:
 - HOLDFIELD
 - HOLD-FIELD
 - HOLD_FIELD
 - TIMESTAMP
 - TIME-STAMP
 - TIME_STAMP
 - LOGCOUNTER
 - LOG-COUNTER
 - LOG_COUNTER

and the field meets the format criteria for a hold field, Natural Construct uses this field as the logical hold field.

Default Business Object Description

Specify a default business object description by assigning a name to the file's Literal Name attribute in Predict. This name is defaulted as the business object description when using a super model. Additionally, this name is displayed when the file is referenced in error messages.

To define a default value for the object description, specify a default value in the Literal Name field for the file in Predict.

Descriptive Browse Fields

When the user invokes a browse attached to a field on a maintenance form, it is referred to as a foreign field browse. When invoked, a foreign field browse displays only the foreign field values unless you designate other fields in the foreign file as descriptive. For example, suppose you know that the warehouse number field in a warehouse file will be referenced as a foreign field browse on a number of maintenance dialogs or pages. To help users select the correct warehouse when browsing, you can designate another field, such as the Warehouse Name field, as descriptive. When users browse for a warehouse number, the descriptive value (in this case, a warehouse name) is displayed, along with the warehouse number.

A descriptive field is designated in Predict by associating a special keyword with the field. You can indicate that certain fields are descriptive in all situations, while others are descriptive depending on the form or page that contains the foreign field.

For more information about descriptive fields, see **Displaying Descriptions for a Foreign Field**, page 359, *Developing Client/Server Applications* or **Generating and Customizing HTML Templates**, page 105, *Developing Web Applications*.

File Volume Information in Client/Server Applications

You can supply information related to the size and stability of your files in Predict. These values are used to determine the default behavior of standalone browse dialog boxes and browses linked to a maintenance dialog box. For more information about linking browse and maintenance functions, see **Understanding the Browse and Maintenance Integration**, page 343 *Developing Client/Server Applications*.

Creating a Domain and Setting Up Security

The application environment includes users, application libraries, business objects and their associated modules. Users are combined into larger entities known as “groups”. Application libraries, business objects and their associated modules are combined into larger entities known as “domains”. Before creating an application with Construct Spectrum, you must define a domain for the application. Before users can access the application, you must grant access to the business objects and object methods within the domain. Three tasks are involved:

- Define a steplib chain
- Define a domain
- Define security for the domain

These tasks are described in the following sections.

Define a Steplib Chain

The first step in setting up a domain is to define the domain's steplib chain. A steplib chain identifies where your application libraries reside on the server. You must set up a steplib chain and link it to your application domain so that application modules can be located and executed.

When defining your steplib chain, keep the following tips in mind:

- Before adding a steplib entry, determine the database ID (DBID) and file number (FNR) of the FUSER system file that you are using.
- The library in which the Spectrum dispatch service is executing is checked before the steplib chain. Therefore, you do not have to add this library to your steplib chain.
- You do not have to specify a DBID and FNR value for a library if you want to use the default DBID and FNR values from the current FUSER system file at runtime.
- Add your FUSER file in the SYSTEM library to the steplib chain. Most generated applications use server framework components that are supplied with Construct (prefixed with “CD” or “CC”). These components are typically installed in the SYSTEM library on the FUSER system file.
- Any components (for example, subprograms, copycode, and data areas) that are required by your generated methods must be available in your application library or one of its steplibs.
- This version requires Natural V2.3.2 or higher. Both the FUSER and FNAT system libraries are automatically added to your steplib chain.
- Record the name of the steplib chain that you define. Add this steplib chain to the application domain that you will set up in the next section, **Define a Domain**.

Tip: If you are new to Construct Spectrum, set up a sample environment. For example, set up a sample application library and link it to your sample steplib chain. Additionally, use the same name to identify your application library, your steplib chain, and your domain.

➤ To access the Maintain Steplib Table panel:

- 1 Log on to the library SYSSPEC and type “MENU” at the Next prompt.
The Construct Spectrum Administration subsystem main menu appears.
- 2 Enter “AA” in the Function field.
The Application Administration main menu appears.
- 3 Enter “MM” in the Function field.
The Application Administration Maintenance menu appears.

- 4 Enter "ST" in the Function field.
The Maintain Steplib Table panel appears:

```

BSSD__MP   ***** Construct Spectrum Administration Subsystem *****   BSSD__11
Aug 31,99           - Maintain Steplib Table -                               10:55 AM

*Action (A,B,C,D,M,N,P)  _

Steplib Name.....: _____

+-----+-----+-----+
| Library | DB   | FNR |
+-----+-----+-----+
| 1       | _____ | _____ |
| 2       | _____ | _____ |
| 3       | _____ | _____ |
| 4       | _____ | _____ |
| 5       | _____ | _____ |
| 6       | _____ | _____ |
| 7       | _____ | _____ |
| 8       | _____ | _____ |
+-----+-----+-----+

Direct Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---
confm help  retrn quit          flip  pref                                main

```

Maintain Steplib Table Panel

- 5 Add up to eight application libraries to the steplib chain.

Define a Domain

Domains are used to group related business objects and services. Set up security for your applications by linking selected groups of users to your domain. For more information about user groups, domains, and security see **Define Security for the Domain**, page 57.

The same business object can be set up in multiple domains. The services assigned to the object can be different for each domain. For example, if you have a Customer object that is used in two applications, an Accounts Receivable and a Sales application, the Customer object in the Accounts Receivable application probably requires different services than a Customer object in a Sales application. Consider

setting up two domains, one for each application. The services you assign to the Customer object are based on the business requirements addressed by each application.

The following steps explain how to set up a domain and link it to the steplib chain explained in the previous section, **Define a Steplib Chain**, page 52. By default, all business objects in the domain are accessed using the same steplib chain. You can, however, override the steplib chain for each business object and object method.

For more information about overriding the steplib chain, see **Overriding the Domain's Steplib Chain**, page 144.

Tip: Specify a steplib chain as high in the application architecture hierarchy as possible. This prevents you from having to specify the steplib chain in many places. If the steplib chain applies to an entire application, place it in the appropriate domain. If the steplib chain applies to one object only, identify it in the header portion of the application service definition. In this way, only exceptions need be specified.

- To access the Maintain Domains Table panel:
- 1 Access the Construct Spectrum Administration Subsystem main menu by logging on to the library SYSSPEC and typing “MENU”.
 - 2 Enter “AA” in the Function field.
The Application Administration main menu appears.
 - 3 Enter “MM” in the Function field.
The Application Administration Maintenance menu appears.

- 4 Enter “DO” in the Function field.
The Maintain Domains Table panel appears:

[illegible]

Maintain Domains Table Panel

Add the domain specifying the steplib chain.

Note: Specifying a steplib chain in the Steplib Chain field is optional. If no steplib chain is defined, the Spectrum dispatch service will attempt to locate the business object from the current execution library and then from the FNAT SYSTEM library.

Once you have set up the domain, link it to the user groups, as explained in the next section.

Define Security for the Domain

To make your application available to users, you must grant them security privileges. To set up security, assign users to groups. Groups identify users who require similar access privileges to your application. You can then grant groups security privileges to your application domain. Granting access to a domain enables users to access the objects and methods within the domain.

Tip: You can postpone this task until after you have created and tested your application. At that time, you can better determine what security privileges should be granted to each group.

For each group that is granted access to a domain, you can further define security privileges by granting access to selected objects and object methods. For example, suppose you have an application domain called “Payroll” that contains all of the objects and methods required for your organization’s payroll application. Two types of users require access to the payroll application: managers and data entry personnel. Managers require access to the entire application, while data entry personnel require access only to input hours, vacation time, sick time, and so on. You can set up one group for the managers and one for the data entry personnel. The Manager group is given access to all objects and methods in the Payroll domain, and the Data Entry group is given access only to those objects and methods required to do their job.

For information about defining users and groups, see **Defining Groups and Users**, page 95, *Construct Spectrum Administrator’s Guide*. For information about defining security for groups and domains, see **Setting Construct Spectrum Security Options**, page 123, *Construct Spectrum Administrator’s Guide*.

FEATURES OF THE ABO AND WEB WIZARDS

This chapter introduces you to the Construct Spectrum Add-In tools you will use to build, customize, and support Spectrum web and ABO projects. These tools include the Configuration Editor that you can use to customize environmental settings and the Regenerate Multiple function that allows you to regenerate many modules. This chapter also describes how you can use implied user exits and the `cst:PRESERVE` tag to protect and preserve your custom code as you generate and regenerate modules. Also explained is the Reports function that you can use to view the actions and changes that occur during a generation or regeneration, and the Cache Viewer from which you can quickly access stored information.


The following topics are covered:

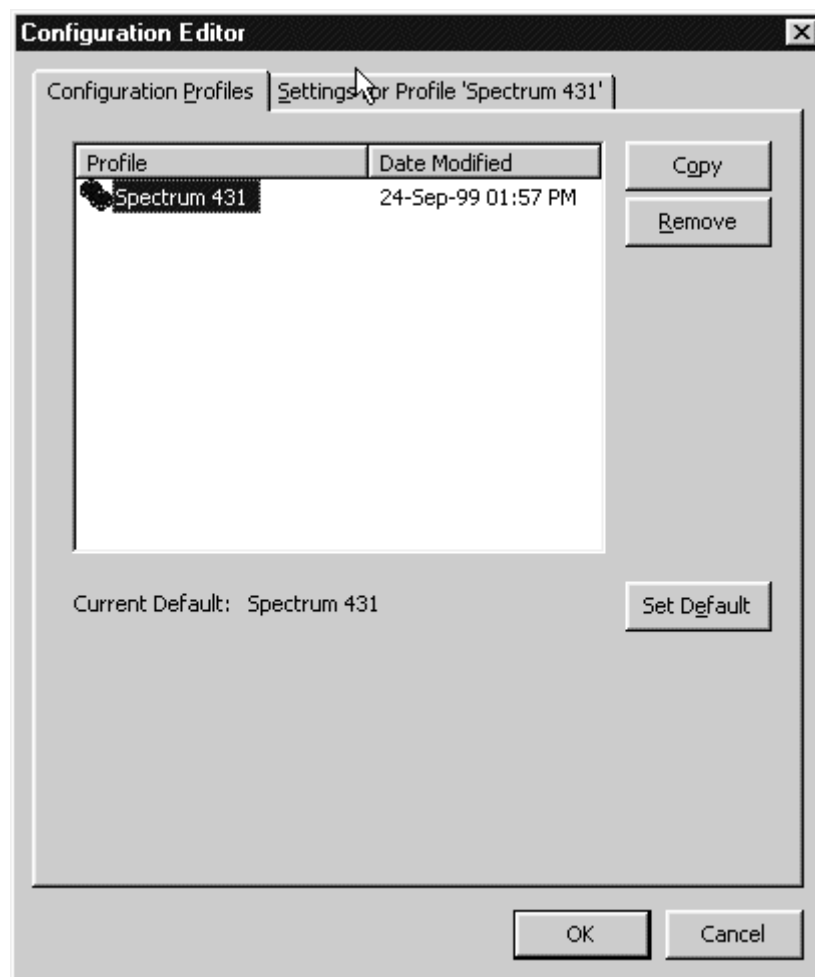
- **The Configuration Editor**, page 60
- **Working with Code**, page 70
- **Regenerating Modules**, page 72
- **Editing Modules**, page 76
- **Using Reports**, page 77
- **Using The Spectrum Cache**, page 83

The Configuration Editor

Use the Configuration Editor to specify global settings such as Spectrum services and generate parameters in your Visual Basic development environment.

Invoking the Configuration Editor

- To invoke the Configuration Editor, perform either of the following actions:
 - From the **Spectrum** menu, select **Configuration**.
 - Or
 - Click  on any wizard step.
- The **Configuration Editor** appears:



Configuration Editor — Configuration Profiles

The **Configuration Profiles** tab displays all available profiles and the date they were last modified. Spectrum 431 is the default profile.

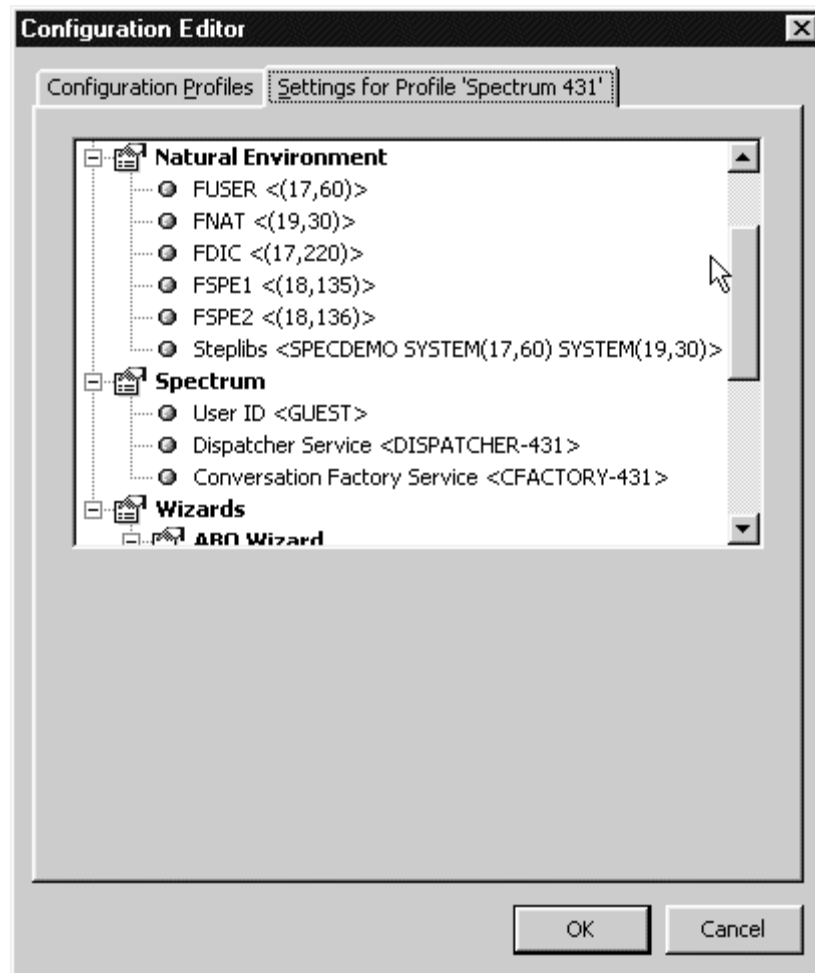
From this tab you can:

- Click **Copy** to make a replica of the current profile, add it to the Profiles list and customize its settings. For more information, see **Copying Profiles**, page 66.
- Click **Remove** to remove a profile from the Configuration Editor. This button is disabled if you have selected the default profile.
- Click **Set Default** to make the selected profile the default.
- Select a profile you want to modify and click the **Settings For Profile** tab to display the settings of the specified profile.

Modifying Profile Settings

The Configuration Editor allows you to view and modify a variety of settings for your Spectrum web or ABO projects.

- To modify profile settings:
- 1 Select the profile you want to change on the **Configuration Profiles** tab.
 - 2 Click the **Settings for Profile** tab.



Configuration Editor — Settings for a Specified Profile

- 3 Select the setting you want to change.
Your options for that setting appear below the setting list.
- 4 Change the settings using the supplied options.
- 5 Click **OK** to save the profile and close the Configuration Editor.

Profile Settings

The following table outlines the settings you can modify:

Setting	Description
Global	
Work Offline	This setting indicates if you are connected to the mainframe. You can use the drop-down list to: <ul style="list-style-type: none">• Select True if you want to work offline.• Select False to allow calls to the server.
Source Compare Command	Provides the command used to invoke your source comparison utility from the Reports dialog.
Source Compare Application	Displays the name of the code comparison application you are using. This name appears in the status bar of the Generate Report dialog when the source comparison utility is running.

Setting	Description (continued)
Natural	
FDIC	<p>This setting lists the libraries where your Natural modules are stored. You can:</p> <ul style="list-style-type: none"> Specify the DBID and FNR values of the Predict, Spectrum secured, Spectrum unsecured, FUSER, and FNAT system files. Change the values displayed if you want to access modules in another file. Click the appropriate buttons to remove or add libraries.
FNAT	
FSPE1	
FSPE2	
FUSER	
Steplibs	<p>This setting lists the libraries where your Natural modules are stored. You can use the direction buttons provided to reorder your libraries.</p>
Spectrum	
User ID	<p>This setting displays your user ID. You can specify a new user ID by typing one into this textbox.</p>
Dispatcher service	<p>This setting displays the dispatch service currently used to access the mainframe server. You can:</p> <ul style="list-style-type: none"> Use the drop-down list to select a dispatch service. Click Service Manger to invoke the Spectrum Service Manager, which you can use to copy, edit, add, delete, and ping services.
Conversation Factory service	<p>This setting displays the conversation factory currently used to access the mainframe server. You can:</p> <ul style="list-style-type: none"> Use the drop-down list to select a conversation factory service. Click Service Manger to invoke the Spectrum Service Manager from which you can use to copy, edit, add, delete, and ping services.

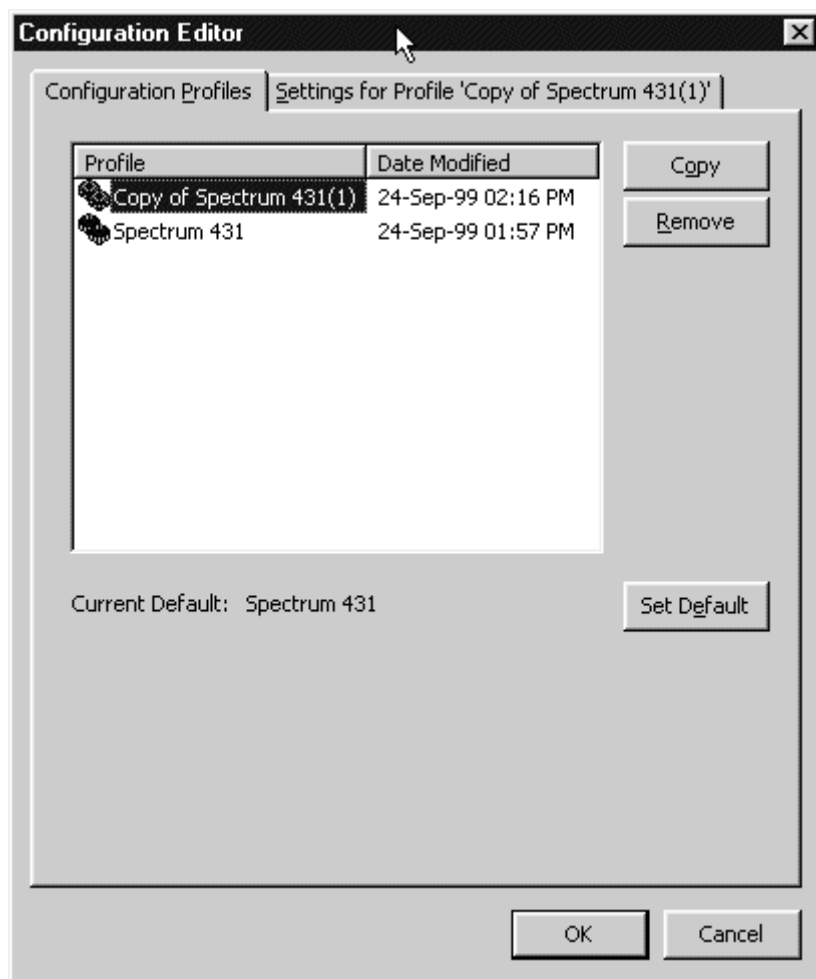
Setting	Description (continued)
Wizards	
ABO Wizard	
Default Arbitrary Class Name	This setting displays the default name for arbitrary subprogram ABO classes you will generate with the ABO wizard.
Default Browse Class Name	This setting displays the default name for the browse ABO classes you generate with the ABO wizard.
Default Maint. Class Name	This setting displays the default name for the maintenance ABO classes you will generate with the ABO wizard.
HTML Wizard	
Default Browse File	This setting displays the default name for the browse HTML templates you will generate using the HTML Template wizard.
Default Maint File	This setting displays the default name for the maintenance HTML templates you will generate using the HTML Template wizard.
Page Handler Wizard	
Default Class Name	This setting displays the default name for the page handler classes you will generate using the Page Handler wizard.

Copying Profiles

You may want to have multiple profiles if you are creating multiple applications with different environmental settings.

➤ To copy a profile:

- 1 On the **Configuration Profiles** tab, select the profile you want to copy.
- 2 Click **Copy**.
A copy of the profile is added to the Configuration Editor as seen below:



Configuration Editor — Copied Profile

- 3 Select the profile, and enter a new name.

- 4 Click the **Settings for Profile <profile name>** tab to specify the configuration settings for the new profile. For information about establishing profile configuration settings, see **Modifying Profile Settings**, page 62.
- 5 Click **OK** to save the profile.

Working with Code

This section describes how you can use implied user exits and the `cst:PRESERVE` tag to protect and preserve your custom code as you generate and regenerate modules.

Implied User Exits

Implied user exits act as placeholders for user exits you have coded after generating a module, ensuring the user exits are placed in the source code in exactly the same order as before you generated the module. However, implied user exits are not added to the generated code unless you have coded them first. Consequently, you must add an exit line to preserve any hand coded changes during the regeneration.

Implied user exits are easily recognized because they use a standard structure and naming convention. Their names are prefixed with the name of the function, subroutine or property you are coding, and suffixed with a location (start or end). For example, you can add two user exits to a function called `Initialize`: `Initialize.Start` and `Initialize.End`. The property procedures also indicate the type of property in the name. For example, a Property Get exit for `CustomerNumber` is: `'CustomerNumber.Get.Start'`

Note: All subroutines have implied user exits at the beginning and end of the routine. For example:

```
Private Sub PerformAction()  
    '<cst:EXIT Name='PerformAction.Start' Implied=True>  
    Dim sval as String  
  
    sval = LookupAction()  
  
    '<cst:EXIT Name='PerformAction.End' Implied=True>  
End Sub
```

Preserving Customizations to Generated Code

Use the `cst:PRESERVE` tag to protect your custom code during a regenerate. Place the tag before and after whole subroutines, entity groups, or between individual variables. For example, to preserve code in the `Class_Initialize` subroutine, add the following code:

```
'<cst:PRESERVE>
Private Class_Initialize()
    Set m_ABO = CreateObject(PROG_ID)
End Sub
'</cst:PRESERVE>
```

Regenerating Modules

This section explains how to regenerate individual and multiple modules. The regeneration process is performed in the background, without your input.

Regenerating Individual Modules

There are two ways to regenerate individual modules.

- To regenerate a single module, either:
 - Right-click the module in the Project Explorer and select **Regenerate** from the shortcut menu.

Or

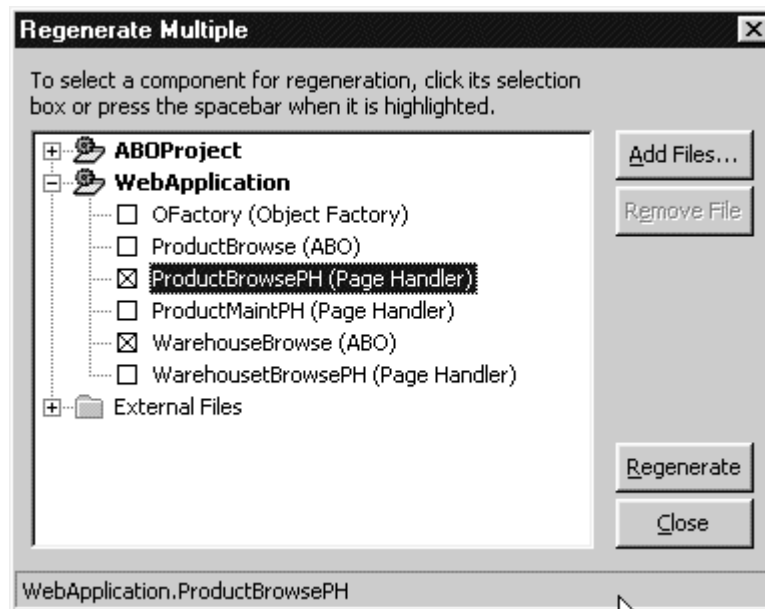
- Select the module in the Project Explorer and select **Regenerate <module name>** from the **Spectrum** menu.

*Note: You can regenerate individual modules by clicking the **Edit <module name> with <Wizard>** from the **Spectrum** menu. This command invokes the wizard you used to generate a module, allowing you to first edit the module specifications and then regenerate it.*


For more information about editing modules, see **Editing Modules**, page 76.

Regenerating Multiple Modules

- To regenerate several modules simultaneously:
 - 1 Open your project in Visual Basic.
 - 2 From the **Spectrum** menu, click **Regenerate Multiple**.
The **Regenerate Multiple** dialog appears:



Regenerate Multiple

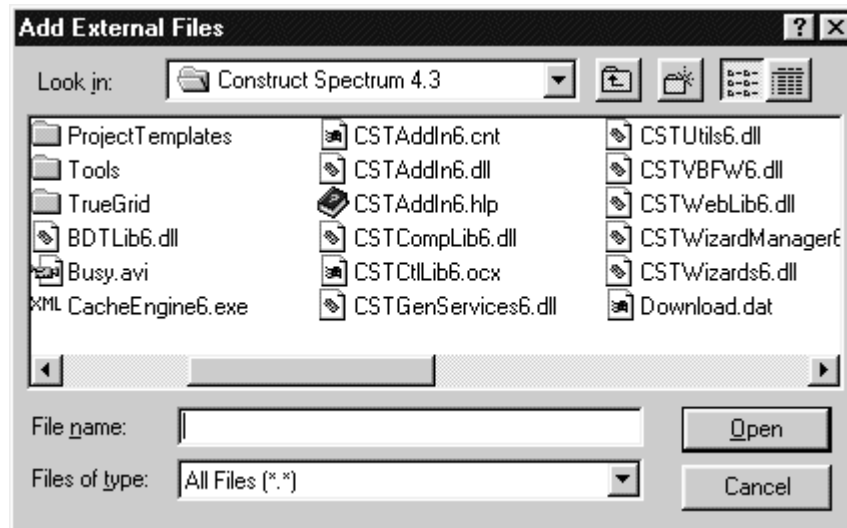
This dialog contains both your project modules and any external files you have added to your projects. This icon  is displayed if you do not have any modules within a project or folder. On this dialog you can:

- Click **Add Files** to move external files into your project for regeneration. For more information, see **Regenerating External Files**, page 74.
 - Click **Remove Files** to move files you do not want to regenerate from this dialog box.
 - Expand the tree to display the individual project modules you want and any external files you want to regenerate.
- 3 Select the modules you want to regenerate and click **Regenerate**.
A message indicates when the regeneration is complete.

Regenerating External Files

Use the **Regenerate Multiple** dialog box to add external files you want to regenerate.

- To add external files:
- 1 Click **Add Files** on the Regenerate Multiple dialog box.
The **Add External Files** dialog box appears:



Add External Files

- 2 Select the file(s) you want to regenerate and click **Open**.
The files are added to the **Regenerate Multiple** dialog box.

Editing Modules

You can invoke the wizard you used to generate a module to edit the module.

- To edit a module, either:
- 1 Select the module you want to modify in the Project Explorer.
 - 2 From the **Spectrum** menu, select **Edit <module name> with <Wizard>**.
The wizard you used to generate the module opens.
 - 3 Modify the model specifications and generate it.

Or

- 1 Right-click the module in the Project Explorer.
- 2 Select **Edit with<Wizard>** from the shortcut menu.
The wizard you used to generate the module opens.
- 3 Modify the model specifications and generate it.

For more information about using the ABO wizard, see **Using the ABO Wizard**, page 117.

For more information about the HTML Template wizard, see **Generating and Customizing HTML Templates**, page 105, *Developing Web Applications*.

For more information about the Object Factory wizard, see **Updating and Customizing the Object Factory**, page 163, *Developing Web Applications*.

For more information about the Page Handler wizard, see **Generating and Customizing Page Handlers**, page 85, *Developing Web Applications*.

Using Reports

This section describes how to generate reports as you generate modules, and how you can review these stored reports as you edit and regenerate modules. It also explains how you can use a code comparison tool to further determine the differences between initial and regenerated code.

Overview

A report is generated every time you use a wizard to generate an ABO or web component. These reports are also stored, allowing you to review the generation process while editing and regenerating modules. If you have a code comparison utility configured to work with Construct Spectrum, you can invoke it from the **Report** dialog to examine code differences between initial and regenerated modules. For more information about using code comparison tools, see **Using Reports with a Code Comparison Tool**, page 82.

Accessing Reports

There are two ways to access reports: as you generate, or by reviewing stored reports.

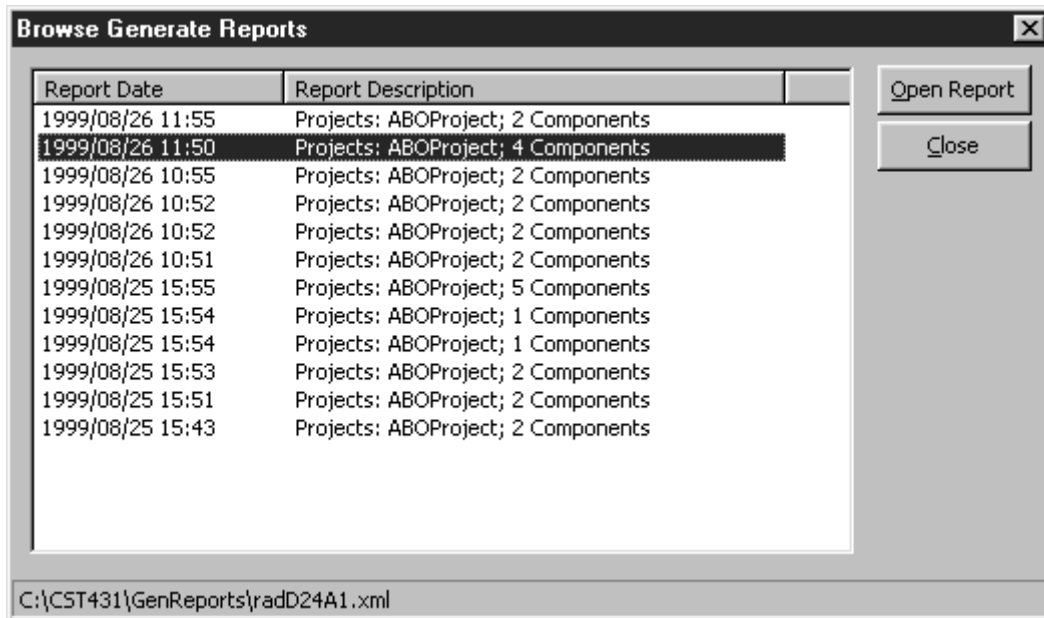
Generating Reports

Clicking **Generate** on the last step of a wizard generates both the module and a report that details the specific actions that occurred during the generation process. This report is automatically stored, allowing you to review it as you modifying and regenerate modules before finally saving them in your Visual Basic project. For more information about the Generate Report, see **Reviewing Stored Reports**, page 78.

Reviewing Stored Reports

It may be useful to review stored reports as you edit and regenerate your application components using the **Generate Report** dialog. You can also invoke your code comparison tool from this dialog to determine the differences between initial and regenerated code. For more information, see **Using Reports with a Code Comparison Tool**, page 82.

- To review a stored report:
- 1 From the **Spectrum** menu, select **Reports**.
The **Browse Generated Reports** dialog box appears:



Browse Generated Reports

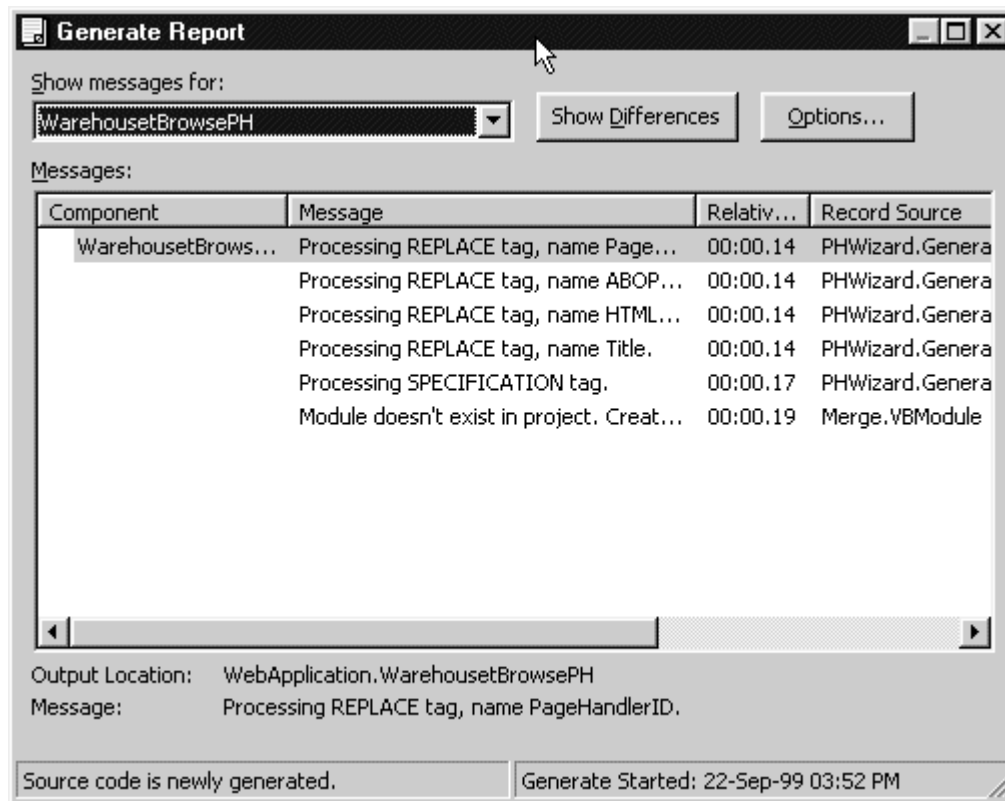
This dialog provides the generation date and the description of saved reports. This includes listing the names of projects in the report and the number of components that were included in the report.

- 2 Select the report you want to open and click **Open Report**.
The **Generate Report** Dialog appears.

For more information about the **Generate Report** dialog, see the following section.

Using the Generate Report Dialog

The **Generate Report** dialog displays any items that were added, removed, or changed, not only in the current module, but also in any other modules affected by the generation, such as LIFDefinitions. It also displays the location of the component, the time the generate was initiated, and any messages.



Generate Report Dialog

Use the **Show Messages For** drop-down list to filter messages according to generate components. The following table outlines the components for which you can view messages:

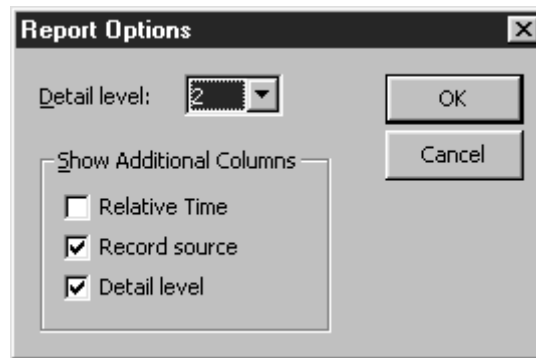
Component	Description
All components	Messages for the generation status of all module components.
System	Generic system process messages that are unrelated to a specific component.
<Specific component>	Messages for the generation of a specific component.

Note: Not all of these components may appear because they are not included in all generated modules.

Specifying Report Detail

You can specify the level of detail you want to see in your report and select other report options and requirements using the **Report Options** dialog box.

- To specify report options:
- 1 On the **Generate Report** dialog box, click **Options**.
The **Report Options** dialog appears:



Report Options Dialog

- 2 Select the options you want.
On this dialog you can:
 - Use the **Detail level** drop-down list to select the level of detail you want the report to display. Level 1 provides a very high level summary, whereas Level 4 provides a highly detailed report.
 - Select **Relative Time** to display exactly when various stages of the generation process occurred.
 - Select **Record source** to view the source of each message.
- 3 Click **OK** to return to the **Generate Report** dialog box.
The dialog contains information in response to all of the options you specified.

Using Reports with a Code Comparison Tool

If you have a code comparison tool that is configured to work with Construct Spectrum, you can click **Show Differences** on the **Generate Report** dialog to view differences between an original and regenerated file. If you do not have a code comparison tool installed or it is not properly configured to work with Construct Spectrum, the **View Differences** button is disabled.

Use the Configuration Editor to configure your code comparison tool with Construct Spectrum.

- To configure the code comparison tool:
- 1 Open the Configuration Editor.
 - 2 Enter a command line in the **Source compare command** textbox, such as:
`"C:\Program Files\BeyondCompare\beyond32.exe" "%1" "%2" /noedit`
 - 3 Click **OK**.
The comparison utility can now be launched by clicking **Show Differences** in the **Reports Generate** dialog.


Using The Spectrum Cache

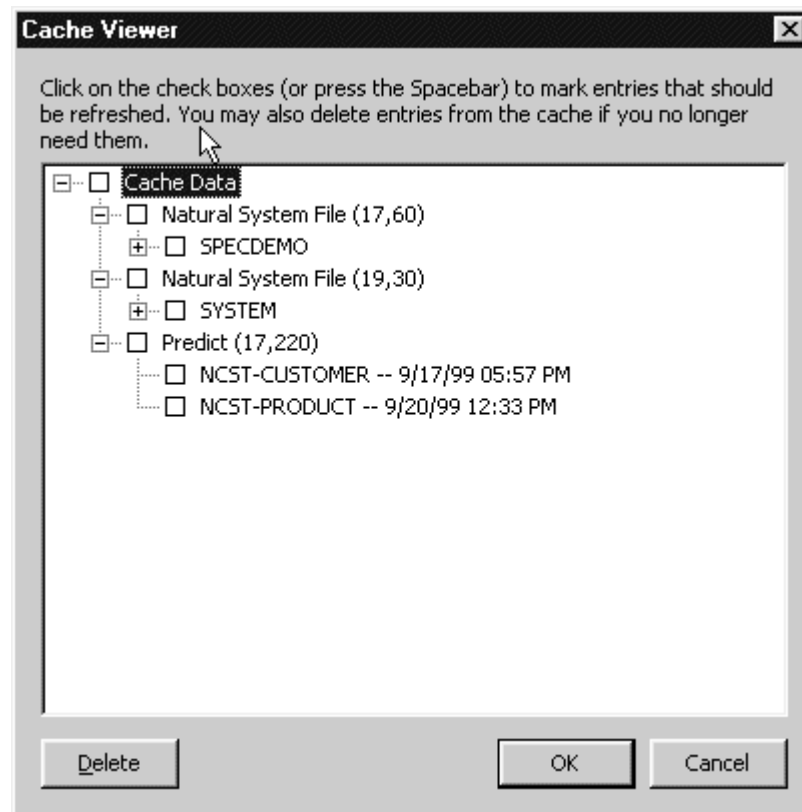
The Spectrum Cache is a dynamic, hierarchical data structure that stores data returned from the server. The cache allows you to quickly store and access values that are used frequently but that take a long time to retrieve or derive. This section explains how to use the Cache Viewer.

Overview

The hierarchical tree structure of the Spectrum cache means it can store complex values such as Predict file definitions. The cache contains all the data used by the wizards during the generate process. It also contains information extracted from FUSER modules and information about the generation environment.

Use the Cache Viewer to:

- Quickly display data in the cache
 - Mark nodes to be refreshed
 - Remove nodes to clean up the cache
- To invoke the Cache Viewer, perform either of the following actions:
- From the **Spectrum** menu, click **View Cache**.
- Or
- Click  on any wizard.
- The **Cache Viewer** appears:



Cache Viewer

The Cache Viewer displays a hierarchical structure of the system files, libraries, nodes, and predict views in your application. The lowest nodes are followed by the date they were last refreshed.

Marking Nodes to be Refreshed

Use the Cache Viewer to select the nodes you want refreshed.

- To mark nodes to be refreshed:
 - 1 Invoke the Cache Viewer.
 - 2 Expand the tree to view individual nodes.
 - 3 Select the node(s) you want to have refreshed.
 - 4 Click **OK**.

When you mark a node, you also mark all of its children. When the wizards fetch data from the cache, they recognize the nodes you specified to be refreshed and make the appropriate call to the server. If the server call fails, existing data in the cache is used.

Removing Nodes From the Spectrum Cache

You can remove nodes that are no longer needed to clean up the cache.

- To remove nodes:
 - 1 Invoke the Cache Viewer.
 - 2 Expand the node tree and select the node(s) you want to delete.
 - 3 Click **Delete**.

The nodes are removed from the cache.

USING THE BUSINESS-OBJECT SUPER MODEL

This chapter explains how to generate multiple Natural components using the Business-Object super model (BUSINESS-OBJECT-SUPER-MODEL). Use this super model to create the Natural components of a Construct Spectrum web application or when you are creating a client/server application without using the Construct Spectrum client framework.

The following topics are covered:

- **Overview**, page 88
- **Before Using the Business-Object Super Model**, page 90
- **Generating Packages Using the Business-Object Super Model**, page 93
- **What to Do if Something Goes Wrong**, page 105

Overview

The Business-Object super model generates all of the required Natural components of a web or other distributable application using a single high-level model specification.

You can generate modules for all of the business objects in your application with one use of the Business-Object super model. A set of modules for a business object is called a “package”. For example, a package for the Customer business object could contain object maintenance and object browse subprograms, their proxies and parameter data areas (PDA).

In order to generate these components, the super model executes separate models. The following table lists the generated modules and the models used to generate them.

Module	Model Name	Result
Object maintenance subprogram, Object PDA, Restricted PDA	Object-Maint-Subp	Subprogram used to maintain a business object. This model also generates the parameter data area and restricted PDA for the object.
Object maintenance subprogram proxy	Subprogram-Proxy	Proxy used to communicate information between the Spectrum dispatch service and an object maintenance subprogram.
Object browse subprogram, Object browse subprogram Key PDA, Row PDA, Restricted PDA	Object-Browse-Subp	Natural subprogram used to encapsulate access to data on the server and return records as a series of rows and columns. The PDAs are used to communicate information to and from an object browse subprogram.
Object browse subprogram proxy	Subprogram-Proxy	Proxy used to communicate information between the Spectrum dispatch service and an object browse subprogram.

Tip: You must generate browse modules for a package if you wish to allow users to browse the business objects in the package. You must also generate browse modules if the business object is linked by a foreign field relationship to another business object. Foreign field relationships enable a user to browse and select key field values for foreign fields on a dialog.

When to Use The Business-Object Super Model

Use the Business-Object super model to generate the Natural modules needed for the first iteration of an application. As you refine your application, you will likely need to regenerate certain application modules. In most cases, you will regenerate these modules separately using individual models.

Tip: The super model does not allow you to specify user exits. To specify user exits, regenerate using the specific model which supports the desired user exit.

Before Using the Business-Object Super Model

This section describes the prerequisite tasks you must perform before generating application packages for a Construct Spectrum application. Before completing any of the tasks described in this chapter, ensure that all required software has been installed and configured on both the server and client.

For information about installing the required software, see the *Construct Spectrum and SDK Client Installation* and the *Construct Spectrum Mainframe Installation Guide*.

Check the Business-Object Super Model Defaults

When the Business-Object super model invokes individual models, it uses the default values determined for each model. Review and change, if necessary, the current defaults for each of these models. To review these values, invoke each model used by the Business-Object super model and note the default values.

Model	Defaults
Object-Browse-Subp	Uses the first four characters of the module name to suffix the object, key, and restricted PDAs.
Object-Maint-Subp	Uses the first four characters of the module name to suffix the object and restricted PDAs.

Set up Default Values in Predict

The Business-Object super model generates specifications for all of the models used to generate an application from a small set of input parameters. To accomplish this, it relies heavily on parameter defaulting. You can add keywords to your file and field definitions in Predict to default various parameters. Customize parameter defaults by linking Predict keywords and verification rules to your fields, files, and relationships.

For more information about Predict defaults and definitions, see **Setting Up Predict Definitions**, page 47.

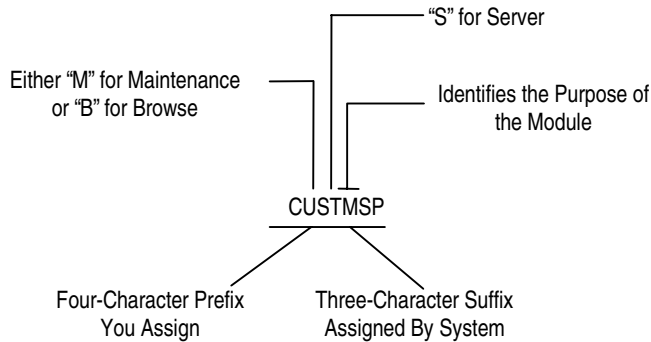
Establish a Naming Convention

Establishing a naming convention is important because multiple modules can be simultaneously created with the Business-Object super model. Identifying modules is easier when your naming convention clearly marks them.

When you use the Business-Object super model, all the modules belonging to a package are given the four-character prefix you assign. If you assign a prefix that is less than four characters, it is padded with dashes. The module name suffix is defaulted by the Business-Object super model. The suffix identifies the module type and can be up to four characters in length. Each suffix name that the Business-Object super model uses as a default is described in the following table.

Default Module Suffix	Module Description
MSO	The object maintenance subprogram and two parameter data areas: Object PDA and Restricted PDA
MSA	
MSR	
MSP	The object maintenance subprogram's proxy
BSO	The object browse subprogram and three parameter data areas: Row PDA, Key PDA, and Restricted PDA.
BROW	
BKEY	
BPRI	
BSP	The object browse subprogram's proxy

The default naming conventions applied to the module name are illustrated in the following diagram.



Naming Convention for a Generated Module

Set Up the Application Environment

Before creating a Construct Spectrum application, you must set up and configure the application environment. This involves the following tasks:

- Define a steplib chain
- Define the domain name
- Define domain security

For more information about these tasks, see **Setting up Your Application Environment on the Mainframe**, page 45.

Generating Packages Using the Business-Object Super Model

You can use either the Construct Windows interface or the Construct Generation subsystem to generate packages using the Business-Object super model. The parameter information you are asked to specify is the same in both interfaces. Each interface also has the same number of input specification steps. In the Generation subsystem, the Business-Object super model has three specification panels: Standard Parameters, Package Parameters, and Specific Package Parameters. Similarly, in the Construct Windows interface, the Business-Object super model wizard uses three steps in which you can specify standard parameters, packages parameters, and new package parameters.

The following sections describe how to use the Business-Object super model to create application packages from both interfaces, using examples from the Construct Windows interface to illustrate these steps.

Generating application packages using the Business-Object super model involves the following steps:

- Defining standard parameters
- Defining general package parameters
- Defining specific package parameters
- Generating the modules

For information about invoking a model, see **Creating a New Specification**, page 98, *Natural Construct Generation User's Manual* and **Using the Super Model in the Generation Subsystem**, page 114, *Developing Client/Server Applications*.

Step 1 — Define Standard Parameters

The **Standard Parameters** step is similar for all model wizards. The parameters in this step are described in **Standard Parameters Wizard Step**, page 269, *Natural Construct Generation User's Manual*.

The screenshot shows a Windows-style dialog box titled "BUSINESS-OBJECT-SUPER-MODEL Wizard". On the left is a dark sidebar with four steps: "Start", "Standard Parameters" (highlighted with a light gray square), "Packages", and "Finish". The "Standard Parameters" step is active. The main area is titled "Standard Parameters" and contains the following fields:

- Module:** A text box containing "EHCU" with a tooltip that says "(SPECDEMO on 1000,1002)".
- System:** A text box containing "SPECDEMO".
- Title:** A text box containing "Super Spec for my module".
- Description:** A multi-line text box containing "Specification".
- Message numbers:** A checkbox that is currently unchecked.

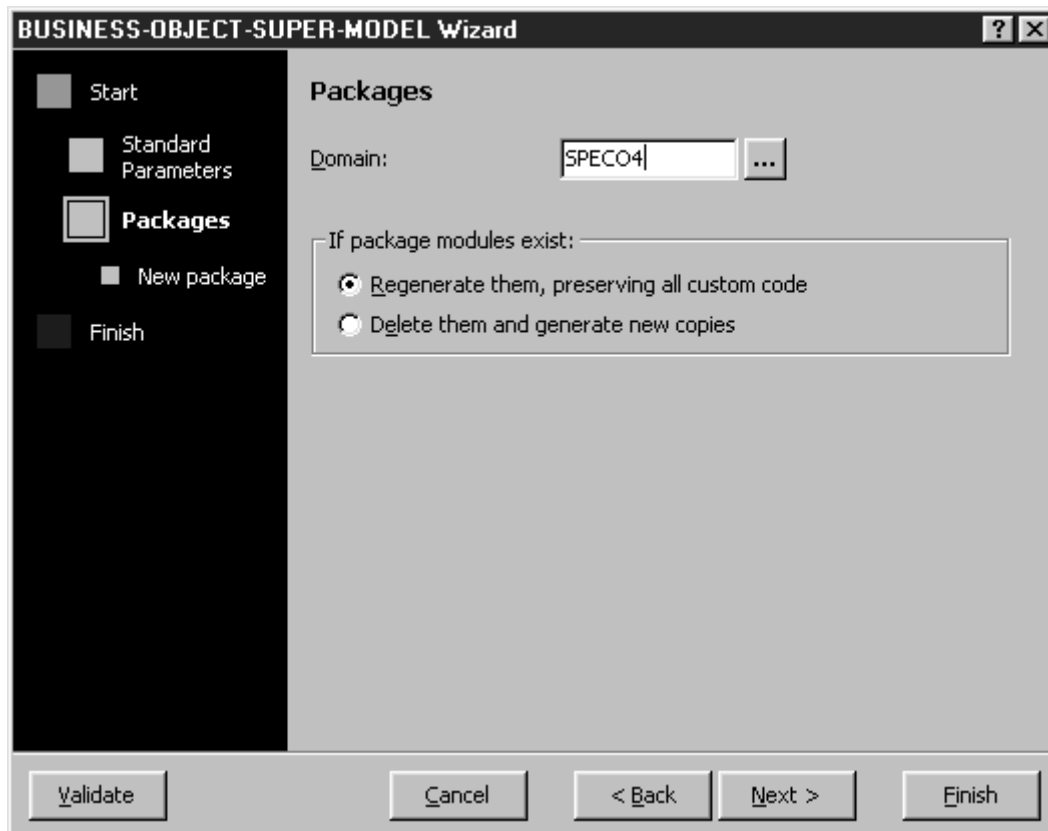
At the bottom of the dialog are five buttons: "Validate", "Cancel", "< Back", "Next >", and "Finish".

Business-Object Super Model Wizard— Standard Parameters Step

- To define standard parameters:
- 1 Type a name for the Business-Object super model specification in the Module field or textbox.
This name identifies the Business-Object super model specification that you are about to create. The name should be descriptive so that you can easily identify it as the Business-Object super model specification for the particular application you are creating.
 - 2 Provide the name of the library or the library identification number where you want to generate the module in the System textbox of field.
By default the current library is displayed.
 - 3 Provide a brief title for the module in the Title field or textbox.
 - 4 Provide a brief description of the model specification in the Description field or textbox.
 - 5 Click **Next** (or press Enter or PF11(Forward)) to specify general package parameters.
The **Packages** step (General Package Parameters panel) appears.

Step 2—Define General Package Parameters

On the **Packages** step (General Package Parameters panel), identify the application packages, up to 12, for which you wish to generate modules. You will also use this step to identify some basic package criteria that affect the entire application.



Business-Object Super Model Wizard — Packages Step

- To define general package parameters:
- 1 Provide the domain name for this application in the Domain field or textbox.
To display a list of domains from which to select a value, do one of the following:
 - In the Construct Window interface, click **...** to view a list of domains that are available. Select a domain and click **OK**.

Or

- In the Generation subsystem, place the cursor in the field and press PF1 to list and select a domain.
- 2 Decide if you want to delete or regenerate existing modules. You have the following options:
 - By default, **Regenerate it, preserving all custom code** is selected. Select this option to regenerate existing modules and save all custom code. When you regenerate modules, any modified parameters in the specification will not be used during the regeneration. However, the Business-Object super model will:
 - Keep user exits
 - Apply updates from information in Predict (such as a new field or a BDT keyword)
 - Apply updates that have been added to the model's code frames
 - If you wish to replace all existing modules with newly generated ones, select **Delete it and generate a new copy**.
 - 3 Click **Next** (or press P11) to invoke the **New Package** step (Specific Package panel) to create a new package.

Step 3 — Define Specific Package Parameters

In this step, you will enter parameters for individual packages. In the Construct Windows interface, all of the packages in your application are represented in the Model wizard navigator. To navigate between packages in the Construct Windows interface, click **Next**, **Back** or select individual packages in the Wizard navigator.

BUSINESS-OBJECT-SUPER-MODEL Wizard [?] [X]

☐ Start
☐ Standard Parameters
☐ Packages
☒ NCST-CUSTOMER
☐ Finish

Package prefix:

PREDICT view: ...

Primary key: ...

Hold field: ...

Description:

Package modules:

Module	Gen.	Model	G/R/O	Library
EHCUMSO	<input type="checkbox"/>	Object Maintenance Subprogram	?	?
EHCUMSP	<input type="checkbox"/>	Spectrum Maintenance Proxy	?	?
EHCUBSO	<input type="checkbox"/>	Object Browse Subprogram	?	?
EHCUBSP	<input type="checkbox"/>	Spectrum Browse Proxy	?	?

Business-Object Super Model Wizard—New Package Step

In the Generation subsystem, press PF8 (frwr) and PF7 (bkwr) to navigate to specific packages. You can also enter a package number in the field following the two angle brackets (>>) to display that package.

- To define specific package parameters:
 - 1 Specify the Package prefix in the Package prefix textbox or field. For more information, see **Establish a Naming Convention**, page 91.
 - 2 Specify the Predict view used by the browse and maintenance subprograms in the Predict view textbox or field.
This view determines which business object will be used.
In the Construct Windows interface, click to view all available Predict files.
Click **Defaults** to retrieve the defaults for the object.
In the Generation subsystem, press PF1 to view all available Predict files.
 - 3 Specify the Primary key in the Primary key textbox or field.
The key can be a descriptor, superdescriptor, or subdescriptor. If the key does not exist in the corresponding Predict file, an error message is displayed upon validation. This value can't be the same as that in the Hold field.
In the Construct Windows interface, click to view the available files for this text box.
In the Generation subsystem, press PF1 to view the available files for this field.
 - 4 Specify the Hold field, the name of the field used to logically protect the record against intervening Update or Delete actions in the Hold textbox or field.
In the Construct Windows interface, click to view the available files for this text box.
In the Generation subsystem, press PF1 to view the available files for this field.
 - 5 Provide a brief description of your package file in the Description textbox or field.

Tip: Based on how the file is defined in Predict, the Business-Object super model attempts to provide default values for these three fields. You can also specify your own default override values using Predict keywords. Rather than typing these values directly, set up your file definition in Predict to default the required values. For more information, see **Setting Up Predict Definitions**, page 47.

- 6 Select the package modules you wish to generate.
In the Generation subsystem, press PF5 to select all the modules.

To select all the modules in the Construct Windows interface, right-click and select **Select All Modules** from the shortcut menu. The **Package modules** grid contains the following information:

Column	Description
Module	All of the modules that can be generated with the Business-Object super model are listed. Each module is identified by the package prefix, followed by the standard suffix for the module type. For more information about suffixes, see Establish a Naming Convention , page 91.
Gen.	Use the Generate check boxes to specify which modules will be generated.
Model	The individual models that the Business-Object super model invokes to generate the package modules. This includes the Object-Browse-Subp model, the Object-Maint-Subp model along with the proxies and PDAs they generate.
G/R/O	<p>“G” indicates that modules do not currently exist in source form and will be generated and saved in the current library.</p> <p>“R” indicates that modules currently exist in source form and will be regenerated and saved in the current library. This status occurs when you select Regenerate it, preserving custom code in Step 2.</p>

Column	Description (continued)
	“O” indicates that modules currently exist in source form and will be overwritten and saved in the current library. This status occurs when you select Delete it, and generate a new copy in Step 2.
Library	<p>Displays any of the following information:</p> <ul style="list-style-type: none"> • A question mark (?) indicates that you must click Check to determine if there is existing source or compiled (object) code for the module. • No content indicates that a check has been made, but there is no existing code for the module. • “S” indicates that source code exists. If the “S” is black, the source code is in the current library. If the “S” is red, the source code is in another library. To view the location of the source code, place the mouse pointer over the “S.” A pop-up window shows the library or libraries. • “C” indicates that compiled (object) code exists. If the “C” is black, the source code is in the current library. If the “C” is red, the source code is in another library. To view the location of the source code, place the mouse pointer over the “C.” A pop-up window shows the library or libraries.


Step 4 — Creating Another Package (Optional)

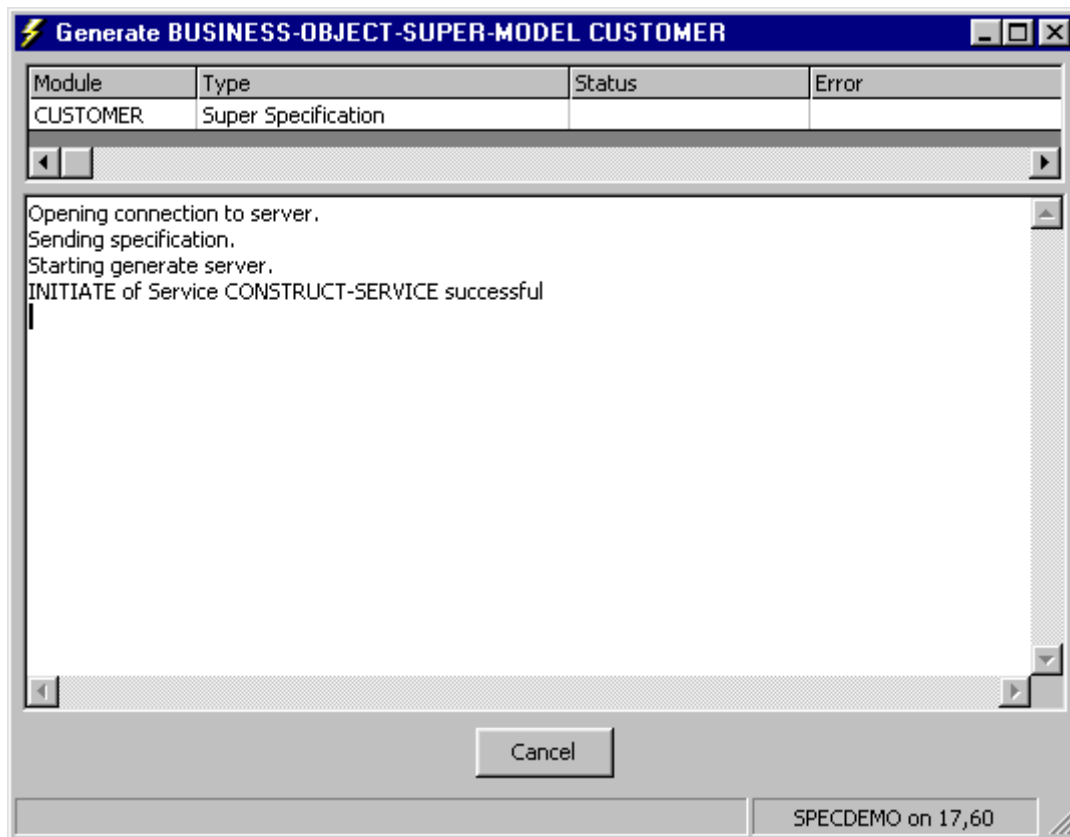
You can create another package from this wizard step.

- To create another package, perform one of the following actions:
 - In the Construct Windows interface, click **Next** or **Add** to create another package. Complete the package as describes in steps 1, 2, and 3.
- Or
 - In the Generation subsystem, press Enter to create a new package. Complete the panel as described in steps 1, 2 and 3. When you have entered specifications for all of your packages, save the Business-Object super model specification and return to the Natural Construct Generation Main menu.

Step 5 — Generating the Modules

Generating Modules in the Construct Windows interface

- To generate the modules:
- 1 Click **Finish** in the Package step.
The Code window appears.
 - 2 From the **File** menu, select **Generate**, or click  on the toolbar.
The **Generate** window appears displaying the generation process:



Generate window

The module status pane displays the modules as they are generated and stowed by the Business-Object super model. The messages pane provides a status report of the generation process, including any error messages that may occur. When all of the modules have been generated and stowed, a generation confirmation message appears in the messages pane.

Note: Click **Cancel** to terminate the generation process at any time.

Generating Modules in the Generation Subsystem

In the Generation subsystem, you can either generate in batch or you can generate from the main menu. In the Construct Windows interface generation is automatically done in batch.

Tip: If you are generating a number of modules, generate in batch to avoid tying up system resources.

- To generate the modules from the main menu:

If you have just completed Step 5, the specification is still in the Natural Construct edit buffer. If the specification is not present, read it into Natural Construct and proceed.

- Enter “G” in the Function field.
The Business-Object super model specification is saved. All the specifications for the individual modules are created and saved.

- To generate modules in a batch:

- 1 Save the specification from the Natural Construct Generation main menu.
- 2 Use the NCSTBGEN utility in batch to generate, specifying the name of your Business Object super model specification, and the model name: BUSINESS-OBJECT-SUPER-MODEL. For information about using this utility, see **Multiple Generation Utility**, page 984, *Natural Construct Generation User's Manual*.

What to Do if Something Goes Wrong

After generating with the Business-Object super model, you should review the generation status report to reconcile any errors that may have occurred. If a module was generated but not stowed because of a missing DDM, for example, you can regenerate the missing modules at a later time after the error has been corrected.

If there was a generation error for a specific module because of a missing dependent module, for example, you can regenerate the individual module from its model specification after you have corrected the error. If the generation errors affect several of the individual modules, you may find it easier to regenerate them from the original Business-Object super model specification after you correct the error. Reread the original Business-Object super model specification into Construct Spectrum and mark only those modules that require regeneration. Then repeat the generation step until all the modules have been successfully generated and stowed.

Note: Ensure that the SYNERR parameter is set to “ON” in your user profile’s NATPARM. Otherwise, compilation errors in the code generated by the Business-Object super model may cause cycling.

USING ACTIVEX BUSINESS OBJECTS

This chapter provides information on ActiveX Business Objects (ABOs). An ABO is a Visual Basic class that wraps the Spectrum calls required to communicate with a Natural subprogram exposed by a subprogram proxy. The class exposes a set of interfaces that make using the subprogram easy. This chapter also contains information on how to use the ABO Project wizard and the ABO wizard. It also explains how to customize the ABOs you generate.

The following topics are covered:

- **Overview**, page 108
- **Prerequisites**, page 109
- **Using the ABO Project Wizard**, page 110
- **Using the ABO Wizard**, page 117
- **Customizing the ABO**, page 124

Overview

Using ABOs is advantageous as they provide a consistent and familiar interface to Natural components. ActiveX business objects are generated by a wizard that is installed as a Visual Basic Add-In. This wizard can be invoked to generate a new business object, regenerate an existing business object, or examine the specifications of an ABO.

This chapter explains how to use the ABO Project wizard to create a project to contain the ABOs you will generate. Once you have created the project, you can use the ABO wizard to generate ABOs.

Prerequisites

Before you can generate an ABO, the back-end Natural modules such as subprograms and subprogram proxies must be generated. For more information, see **Generating a Subprogram Proxy**, page 132.

Ensure that you know the names of the subprogram proxies and their location.

Using the ABO Project Wizard

The following section explains how to use the ABO Project wizard to create the ABO project. It also explains the framework components that Construct Spectrum adds to the project.

Creating the ABO Project

➤ To create the ABO Project:

- 1 Start Visual Basic

The **New Project** dialog appears:



New Project

- 2 Select **Spectrum ABO Project** and click **Open**.

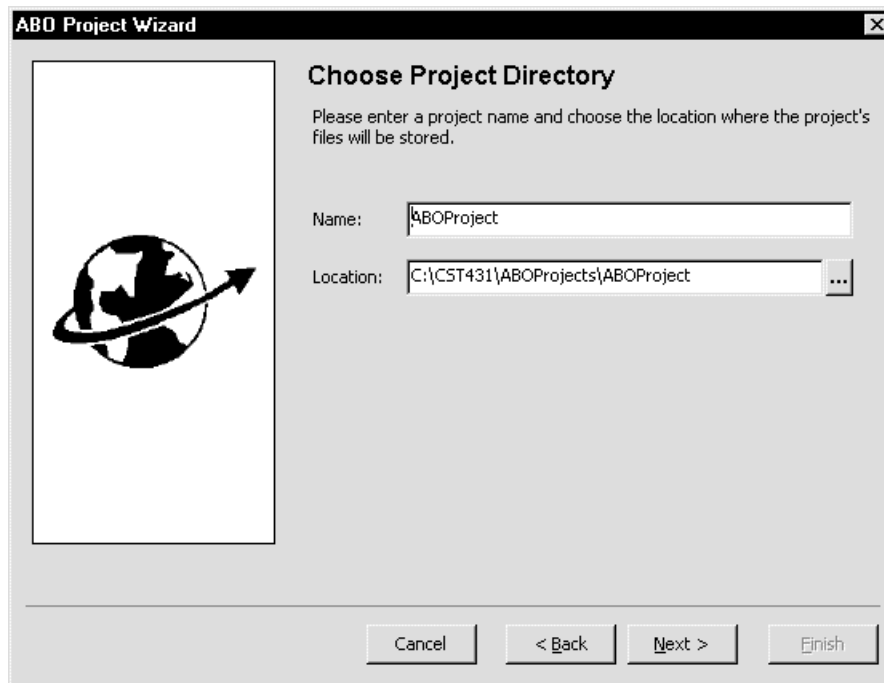
The ABO Project wizard appears:



ABO Project Wizard

- 3 Click **Next**.

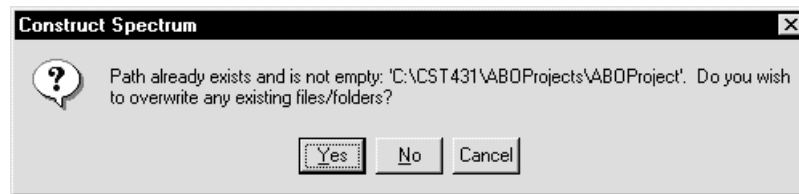
The **Choose Project Directory** dialog appears:



Choose Project Directory

- 4 Enter your project name and the location where you would like to store your project. We recommend that you store the ABO project in the same directory where your web applications will be stored.
- 5 Click **Next**.

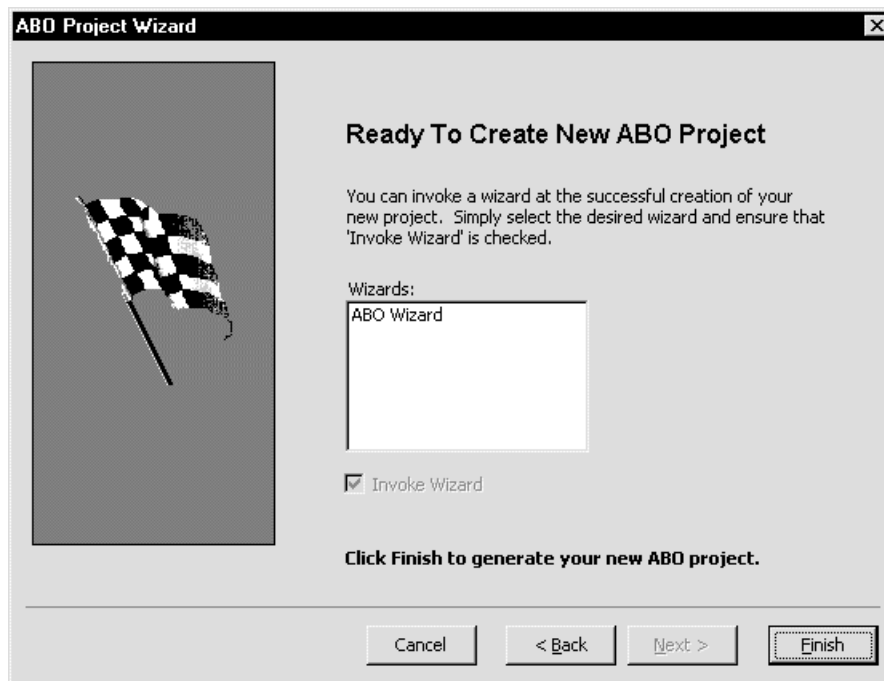
If you keep the default directory or enter another directory that does not exist, you will see the following window:



Create New Directory

- 6 Click **Yes** to create the new directory.

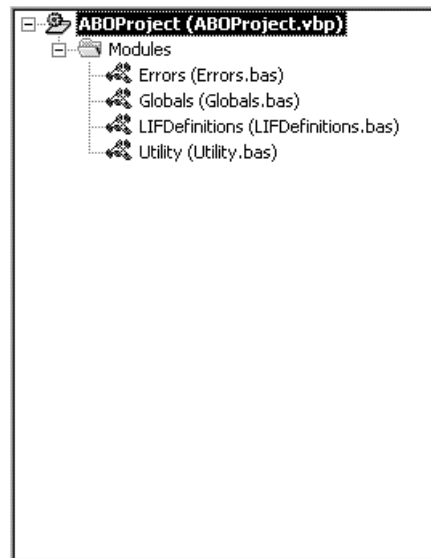
The Ready to Create New ABO Project dialog automatically appears:



ABO Project Wizard — Ready To Create New ABO Project

- 7 If you wish to invoke the ABO wizard after creating the project, select the wizard and click **Invoke Wizard**.
- 8 Click **Finish**.

The ABO project appears in Project Explorer:



ABO Project in Visual Basic

Framework Components of the ABO Project

The following table describes the framework components that Construct Spectrum adds to the ABO project.

Module name	Description
Errors.bas	This module provides error-raising capabilities for the ActiveX component.
Globals.bas	This module contains definitions, variables, and helper routines used by the generated ABOs.
LIFDefinitions.bas	This file is empty in a new ABO project. It becomes populated with Natural data area definitions when ABOs are added to the project using the ABO wizard.
Utility.bas	<p>This module contains procedures that could be used in any type of Visual Basic application.</p> <p>For example, Subst performs substitution of values into a string and is useful for developing international applications. For example:</p> <pre>'This is the message that contains substitution placeholders. 'This message might come from a resource file in the case of a localized app. smsg = "Value must be in the range %1 to %2." MsgBox Subst(smsg, 100, 10000), vbExclamation</pre>

Using the ABO Wizard

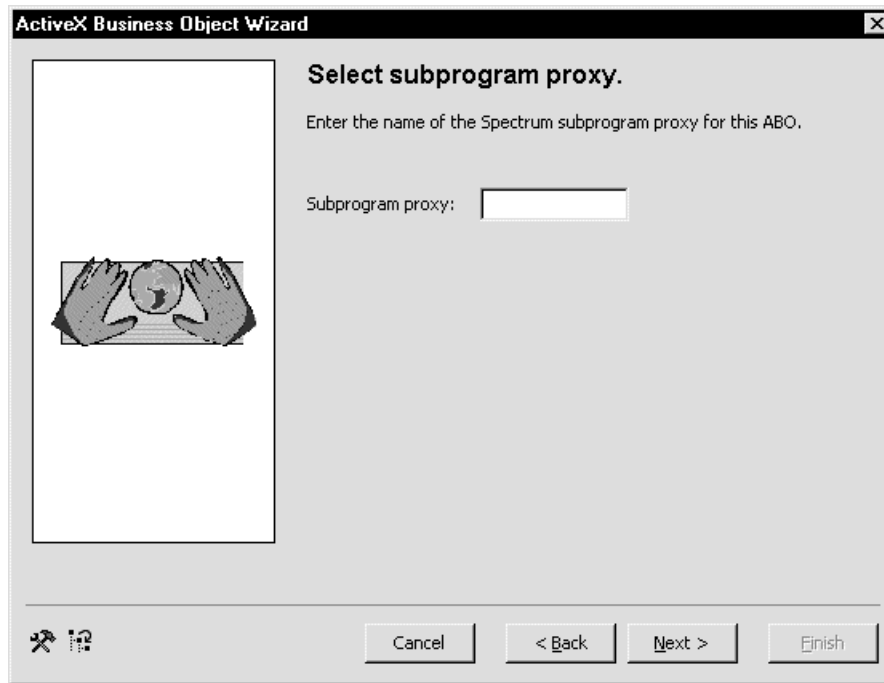
After you have created an ABO project, the next step is to generate an ABO for each of the Natural subprograms in your application.

- To generate an ABO and add it to the project:
- 1 On the **Spectrum** menu, point to **Wizards** and select **ActiveX Business Objects**. The ActiveX Business Object Wizard appears:





ActiveX Business Object Wizard

- 2 Click **Next**. The Select Subprogram Proxy step appears:



ABO Wizard — Select Subprogram Proxy

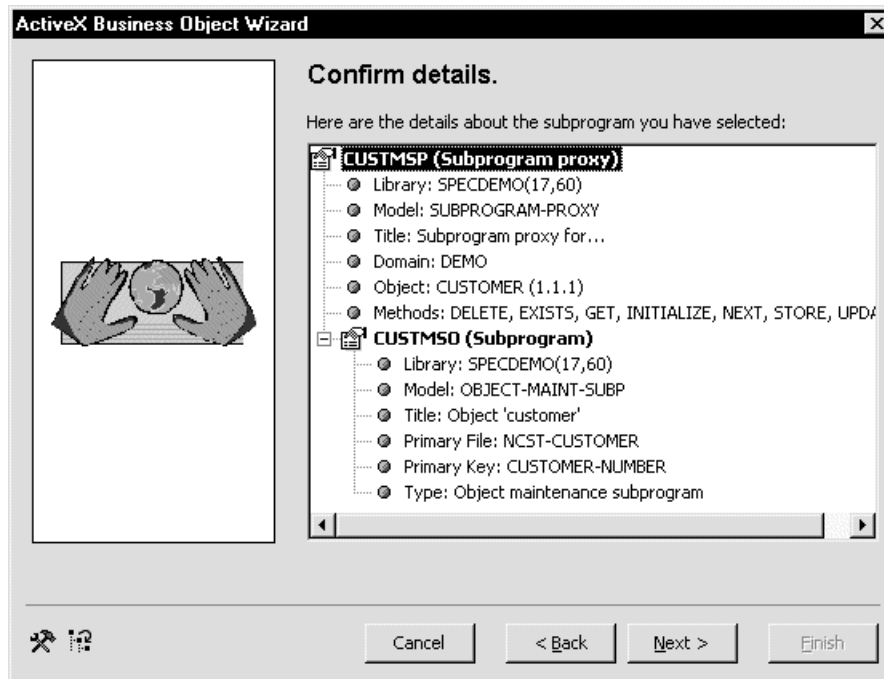
Note: For information about using the Spectrum Cache viewer  or Configuration editor , see **Features of the ABO and Web Wizards**, page 59.

- 3 Enter the name of the subprogram proxy for the ABO.
- 4 Click **Next**.

The wizard performs the following steps:

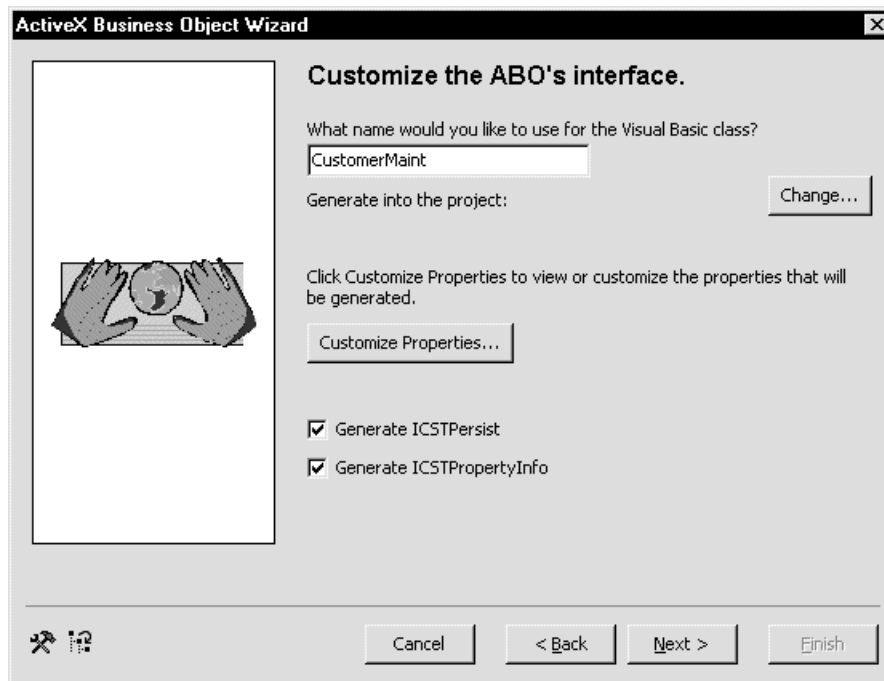
- It reads the ****SAG** lines of the subprogram to extract the model name of the subprogram.
- For an object browse subprogram, it reads the logical key names.
- It reads the ****SAG** lines of the subprogram proxy to extract the domain, object, version, subprogram name, and 1:V overrides.
- It accesses the Spectrum files to retrieve the method names linked to this subprogram proxy.

The Confirm Details step appears:



ABO Wizard — Confirm Details

- 5 This step shows the library, model name, title, domain, object, and methods for the subprogram proxy, as well as information about the proxy's subprogram. You can confirm that everything is correct, with the option to click **Back** if you wish to select a different proxy.
- 6 Click **Next**.
The Customize the ABO's Interface step appears:



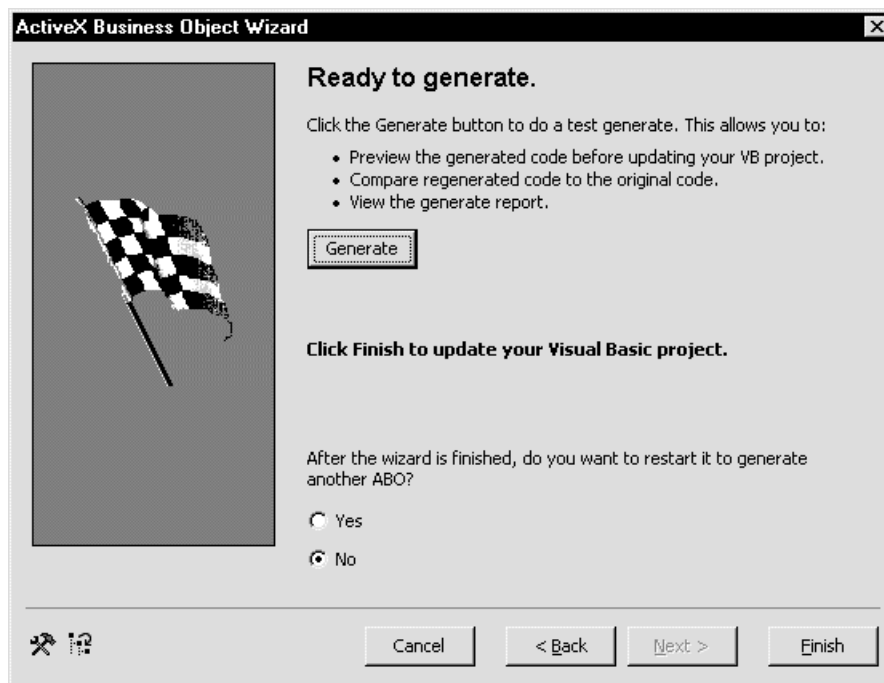
ABO Wizard - Customize the ABO's Interface

- 7 Check the default name supplied for the ABO and change it if you wish.
- 8 Check that the ABO will be generated into the correct project. If you wish to change the project, click **Change** and select a project from the dialog that appears.
- 9 If you wish to customize the ABO's properties, click **Customize Properties**. For more information, see **Customizing the ABO**, page 124.

- 10 Check the status of the **ICSTPersist** option, which allows the ABO to save all of its instance data at runtime and then restore that data at a later time.
- 11 Check the status of the **ICSTPropertyInfo** option, which provides extended information about the properties exposed by the ABO. This information can be accessed at run-time. The extended information includes:
 - property name
 - VB data type
 - number of decimal places for numeric property
 - length of the data
 - logical format
 - read-only or not read-only
 - number of dimensions in an array
 - number of occurrences in each dimension

12 Click **Next**.

The Ready to Generate step appears:



Ready to Generate

At this point, you can perform the following actions:

- If you wish to view the generation report, click **Generate**.
If you have a code comparison utility installed and configured for use with Construct Spectrum, you can also compare the new code you generated with code from an earlier generation of the module.

For information about using a code comparison utility with Construct Spectrum, see **Using Reports with a Code Comparison Tool**, page 82.

For information about the generation report, see **Using the Generate Report Dialog**, page 79.

- Click **Finish** to complete the generation.
When the generation is complete, a message window informs you of the success or failure of the operation. If there were problems with the generation, the window prompts you to view the generation report.

The wizard gives you the option to generate another ABO before leaving and returning to Visual Basic.

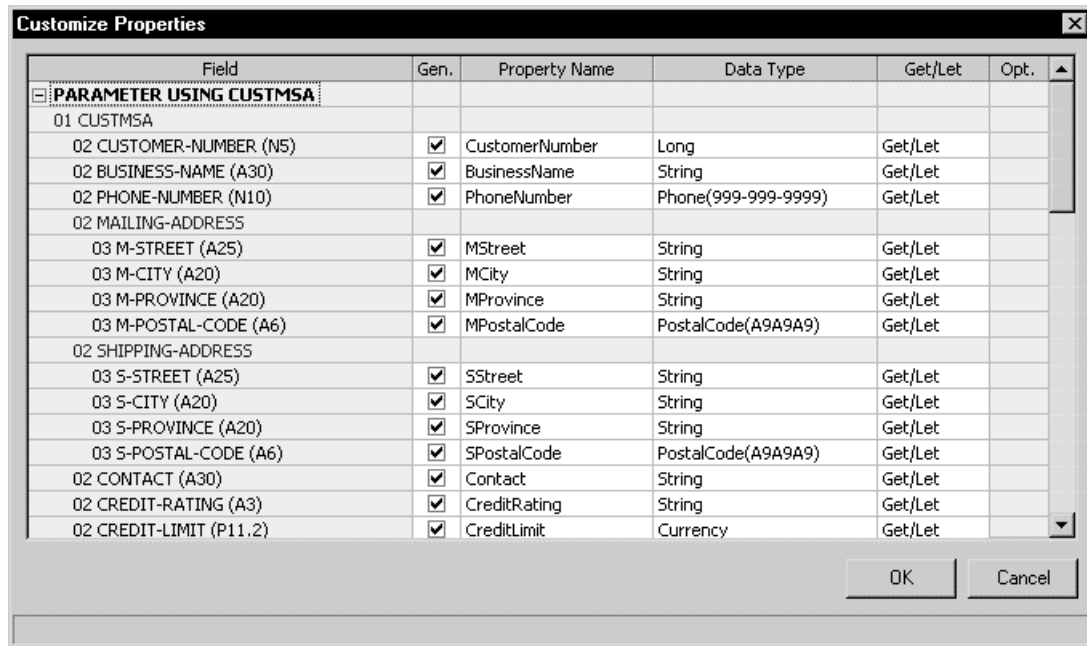
Customizing the ABO

You have the option of customizing columns on the **Customize Properties** dialog.

Customizing the Properties Generated for the ABO

The ABO interface can be customized or simply viewed before generating the ABO.

Click **Customize Properties** in the Customize ABOs Properties step to display the **Customize Properties** dialog:



Customize Properties


By default, if the subprogram is an object maintenance subprogram, properties will be generated for the fields in the object PDA. If the subprogram is an object browse subprogram, properties will be generated for the key PDA and row PDA. All other types of subprograms will have properties generated for the entire PDA.

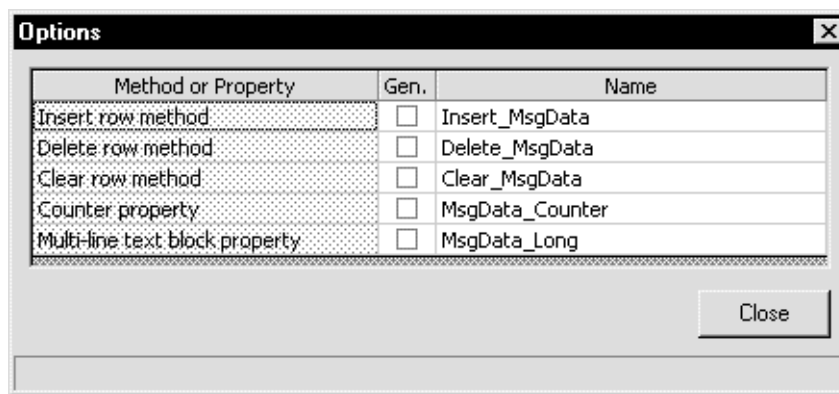
If the subprogram is an object maintenance subprogram, the wizard displays the method names and their generated derivations, of which the method names can be customized. If the subprogram is an object browse subprogram, the wizard displays the logical key names.

You can customize the derived property names, data types, and get/let status (read/write). Using the check boxes in the Generate column, you can specify which properties should be generated. The **Customize Properties** dialog highlights any fields that have been changed from the default. The following table describes these options in more detail:

Column	Description
Field	The name of the field in the Natural data area.
Gen.	You can suppress default properties from being generated by deselecting their check boxes.
Property Name	The name of the property that will be generated for the Natural field.
Data Type	Native Visual Basic data type that the property is declared as.
Get/Let	Get returns the value of a property. Let sets the value of a property.
Opt.	Allows you to specify whether added methods or properties should be generated for the array. Applicable to MUs and PEs only. See the next section for more information.

Using the Opt. Column

- To view additional properties for MUs and PEs:
 - 1 Click the field's Opt. cell to make the  visible and click it. The **Options** dialog appears:



Options Property

Extra properties are generated for the ABO class by default. You have the choice of manually renaming each method or property name. The Multi-line text block property is only available for alphanumeric MUs. This allows users to edit all the elements of an array at one time, in one continuous text string. For more information, see **Customizing HTML Before Generation**, page 123, *Developing Web Applications*.

Customizing With User Exits in the ABO

The following are user exits supplied in the generated ABO.

GetAppService_.SetMethodAndBlocks

Use this exit to override the default method names and block numbers in the GetAppService_ procedure.

ICSTBrowseObject_LogicalKeyInfo.Extra

This exit resides in the Property Get LogicalKeyInfo procedure in the ICSTBrowseObject interface. This procedure is used to provide information at run-time about the logical keys supported by the ABO. Use this exit to define additional logical keys that you have added to the ABO manually.

This exit is only available in browse ABOs.

ICSTPersist_InstanceData.Get.Extra

Use this exit to persist additional module-level variables.

This exit is only available if you generate the ABO with the ICSTPersist interface.

ICSTPersist_InstanceData.Let.Extra

Use this exit to restore the additional module-level variables that you persisted in the ICSTPersist_InstanceData.Get.Extra user exit.

This exit is only available if you generate the ABO with the ICSTPersist interface.

ICSTPropertyInfo_PropertyInfo.Get.Extra

This exit resides in the Property Get PropertyInfo procedure in the ICSTPropertyInfo interface. This procedure is used to provide information at run-time about the properties in the ABO's class. Use this exit to define additional properties that you added to the ABO class manually.

This exit is only available if you generate the ABO with the ICSTPropertyInfo interface.

<CounterPropertyName>.Get.NullList

This is a dynamically generated user exit. Every array counter property procedure that is generated will have this user exit.

Array counter property procedures contain code that determines the number of array occurrences that are used. This code examines each occurrence of the array and checks whether certain fields are empty. If one of these fields is not empty, the code considers the array occurrence to be used.

Use this exit to specify the fields that should be checked and the values that the fields should have to be considered empty. The wizard always generates sample code into this exit consisting of the field names and the empty values. You can change the sample code after generating.

Because coded user exits are always preserved when regenerating, you should delete the existing exit if you want the wizard to regenerate the sample.

USING THE SUBPROGRAM PROXY MODEL

This chapter explains the function of the subprogram proxy, how to generate proxies using the subprogram proxy model, and how to customize your proxy. It also contains information about adding a method to an application service definition, overriding block handling, and versioning.

The major functions of the model are to generate:

- A subprogram proxy that interacts with the Spectrum dispatch service and the target Natural subprogram.
- The application service definition entry needed in the Spectrum Administration subsystem.

The following topics are covered:

- **Overview**, page 130
- **Generating a Subprogram Proxy**, page 132
- **Generating a Subprogram Proxy**, page 132
- **Application Service Definition Methods**, page 141
- **Overriding Block Handling**, page 146
- **Versioning**, page 151
- **Support for Debugging**, page 152

Overview

The Subprogram-Proxy model generates a module that acts as a bridge between the Spectrum dispatch service and a specific subprogram. When a request is initiated from a dialog or a web page to the server (for example, when a user updates a customer record), information is sent from the client to the subprogram proxy on the server. The subprogram proxy then calls the object subprogram that can fulfill the request.

A subprogram proxy is also responsible for converting data between the network transfer format and the native Natural data format used in a subprogram's PDA.

Other features that the subprogram proxy provides are optimized data block handling and the creation of application service definitions.

You must generate a subprogram proxy for each subprogram included in your application. You can create subprogram proxies by generating with the VB-Client-Server-Super-Model, the Subprogram-Proxy model, or the Business-Object super model.

If you are creating a new application or have performed extensive changes to your application file relationships, use the super models to generate your application. The Business-Object super model generates object maintenance and browse subprograms, their PDAs, and subprogram proxies. For more information, see **Using the Business-Object Super Model**, page 87.

The VB-Client-Server-Super-Model generates the same Natural modules as the Business-Object super model, as well as the Visual Basic modules needed by client/server applications. For information about using the Super model, see **Using the Super Model to Generate Applications**, page 139 in *Developing Client Server Applications*.

Typically, you would generate with the Subprogram-Proxy model when you are tailoring an existing application. This chapter explains how to generate a subprogram proxy using the Subprogram-Proxy model.

System Files Containing Spectrum Administration Subsystem Data

The Subprogram-Proxy model requires access to the Spectrum Administration subsystem files. There is one file containing secured data and another containing unsecured data. The Subprogram-Proxy model requires access only to the unsecured data in order to generate a proxy. It uses this data to provide an active help listing for the domain field on its primary input panel. Additionally, this model creates or updates an application service definition for the specified object subprogram.

The Spectrum Administration subsystem file containing unsecured data must be available either through an LFILE designation in your startup of Natural or through the Natural nucleus used in the session in which you are generating (NT-FILE parameter must be specified).

Generating a Subprogram Proxy

This section describes how to generate a subprogram proxy and considerations to be aware of when generating the proxy.

There are four possible steps involved in generating a subprogram proxy:

- Specify Standard Parameters for the subprogram proxy
- Specify the actual size of any 1:V fields in the subprogram's PDA
- Add code to user exits
- Generate the subprogram proxy

Before completing these steps, you should be familiar with the considerations described in the next section.

Considerations for Generating With the Subprogram-Proxy Model

Be aware of the following considerations when using the Subprogram-Proxy model to generate a subprogram proxy:

- **Maintaining one application service definition for each business object**
An application service definition specifies the methods and the subprogram from which each method is executed for a business object. The application service definition is created or updated when you generate a subprogram proxy.

To maintain one application service definition for each business object, ensure that the domain name, object name, and version number are the same when you generate each subprogram proxy for the business object. For example, if you have a Customer business object that has both a maintenance and a browse function, generate one subprogram proxy for the maintenance function and one for the browse function. To ensure that only one application service definition is created for both the maintenance and browse functions, specify the same domain, object, and version when you specify the model parameters for each subprogram proxy.

For more information, see **Application Service Definition Methods**, page 141.

- **1:V Variables**
When generating a proxy for subprograms, pay special attention to specific subprograms that have 1:V variables (such as object browse subprograms). Subprograms allow an arbitrary number of rows (records) to be returned to the client by defining the row parameter using Natural's (1:V) notation. Normally, you would want as many records as possible returned for each server request to minimize the number of calls to the server. However, the more records requested, the longer it takes to satisfy each request. Additionally, be sure you do not specify more occurrences than will fit within the maximum 32K communication area available for each request.

Step 1 — Specify Standard Parameters for the Subprogram Proxy

You can access the Subprogram-Proxy model in the Generation subsystem on the server, or you can use the model wizard in the Construct Windows interface. Below is the **Standard Parameters** step in the model wizard:

The screenshot shows the 'SUBPROGRAM-PROXY Wizard' dialog box. On the left is a vertical sidebar with three steps: 'Start' (selected with a dark square), 'Standard Parameters' (highlighted with a light gray square), and 'Finish' (dark gray square). The main area is titled 'Standard Parameters' and contains the following fields and controls:

- Module:** A text box containing 'CUSTMSP' with '(SPECDEMO on 17,60)' to its right.
- System:** A text box containing 'DEMO'.
- Title:** A text box containing 'Subprogram proxy for Cust'.
- Description:** A text area containing 'This subprogram proxy supports the customer order maintenance system'.
- Subprogram:** A text box containing 'CUSTMSO' followed by an ellipsis button '...'.
- Domain:** A text box containing 'DEMO' followed by an ellipsis button '...'.
- Object name:** A text box containing 'CUSTOMER'.
- Version:** A text box containing '1.1.1'.
- Checkboxes:** Three unchecked checkboxes are listed: 'Generate trace code', 'Compress network data', and 'Encrypt network data'.
- Buttons:** At the bottom are 'Validate', 'Cancel', '< Back', 'Next >', and 'Finish'. There is also an 'Edit 1:V Overrides' button next to the Subprogram field.



SUBPROGRAM-PROXY Wizard — Standard Parameters

The **Standard Parameters** step is similar for all model wizards. The common parameters Module, System, Title, and Description are described in **Standard Parameters Wizard Step**, page 269, *Natural Construct Generation User's Man-*

ual. For general information about the Construct Windows interface, see **Using the Construct Windows Interface**, page 77, *Natural Construct Generation User's Manual*.

Tip: Follow the Construct Spectrum naming conventions and type “MSP” for the last three characters of a maintenance subprogram proxy or “BSP” for the last three characters of a browse subprogram proxy. Using these conventions makes it easier to identify the subprogram proxy.

Provide the following parameters:

Parameter	Description
Subprogram	Specify the name of the subprogram for which the proxy is being generated. For example, if you are generating a subprogram proxy for the Customer Order subprogram, ORDMSO, type “ORDMSO” in the Subprogram text box. Click  to select from a list of available subprograms.
Domain	The domain groups related business objects. You can set up security for your applications by linking selected groups of users to each domain.Specify a domain by typing it in this textbox, or click  to select from a list of available domains.
Object name	Specify a name to identify the business object. For example, a customer information business object could be called “Customer”. For more information, see Versioning , page 151.
Version	Identify the version level of your package. This number consists of three parts: version, release, and SM level.

Note: One application service definition is created for both a maintenance and a browse dialog only if the domain, object, and version values are identical when you complete this panel for their respective subprogram proxies.

Parameter	Description (continued)
Generate trace code	Select the Generate trace code check box only if you are developing an early iteration of your application or if runtime errors are occurring in your application. This option adds code to the subprogram proxy that will help you determine the cause of a parameter format error. If your application is stable, do not generate trace code. This improves the performance of your subprogram proxy and reduces the amount of generated code.
Compress network data	Select the Compress network data check box if the proxy will be transmitting large volumes of data to the client. If you are generating a browse subprogram proxy, you should select data compression if a large volume of data is being sent to the client.
Encrypt network data	Select the Encrypt data check box only if the proxy will be transmitting sensitive data to the client.

*Note: The Compression and Encryption parameters apply only to data sent to the client. If you are creating a client/server applications, you can enable compression and encryption for data sent from the client to the server. Mark the **Compress network data** and **Encrypt network data** check boxes in the **Standard Parameters** step of the VB-Maint-Object model or VB-Browse-Object model (depending on the type of dialog you are creating).*

Step 2 — Specify the Number of Occurrences to be Returned

Specify the maximum number of 1:V arrays that can be returned to the client for each request. A 1:V array can consist of either one-dimensional data, such as a list of repeating values, or two-dimensional data, such as a row of record data.

For maximum efficiency, specify 20 occurrences for each subprogram structure (PDA).

- To specify the maximum number of occurrences to return for each request:
- 1 From the **Standard Parameters** step, click **Edit 1:V Overrides**.
If no fields in the target subprogram use the 1:V notation, a message is presented indicating this. Otherwise, the model automatically determines these values and presents a dialog box listing their names. For example:



Edit 1:V Overrides

Note: If you are using the Subprogram-Proxy model in the Generation subsystem on the server, press PF5 (1:V) in the Standard Parameters panel to access the 1:V Overrides panel.

- 2 Specify the maximum number of occurrences that can be returned to the client with each call to the server.
Click **Refresh** when you wish to update the information by making another call to the server.
- 3 Click **OK** to return to the **Standard Parameters** step.

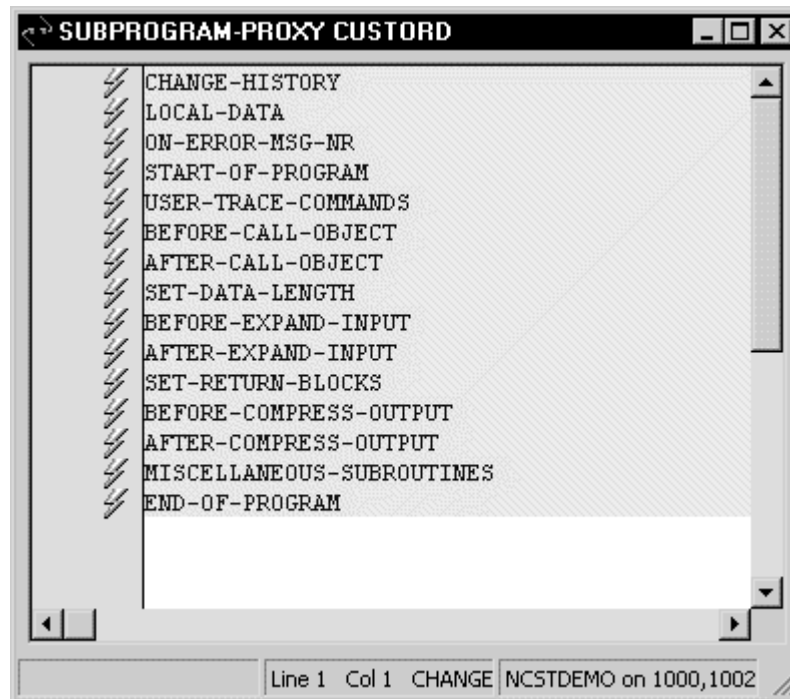
Step 3 — Add User Exits

After supplying model parameters, you can customize the generation results by creating user exit code for the module. Some customizations you should consider are:


- modifying block handling
- adding block handling for new methods

➤ To add user exits:

- 1 From the **Standard Parameters** step, click **Finish**.
The Code window opens, showing the user exits that are available for the Subprogram-Proxy model:



Code Window

The  icon indicates that sample code can be generated for the user exit.

- 2 Right-click the user exit and select **Generate Sample** from the shortcut menu.
- 3 Modify the code as required.

You can also generate sample code from the User Exit list.

➤ To invoke the user exit list, perform any of the following actions:

- From the **View** menu, click **User Exit List**.

Or

- Click .

For more information, see **Using the User Exit List**, page 120, *Natural Construct Generation User's Guide*.

➤ To generate sample code:


- 1 Select the user exit
- 2 Click **Generate sample**.

In this dialog box, you can also add new user exits and insert code into them. For more information, see **Generating User Exit Samples**, page 122, *Natural Construct Generation User's Guide*.

For more information about using the Code window, see **Using the Construct Windows Interface**, page 77, *Natural Construct Generation User's Manual*.

Step 4 — Generate the Subprogram Proxy

➤ To generate the subprogram proxy from the Code window:

- 1 From the **File** menu, click **Generate** or click  on the toolbar.
The Generate window appears, showing module and status information. For more information, see **Generating Modules**, page 124, *Natural Construct Generation User's Manual*.
- 2 When the generation finishes successfully, the subprogram proxy is automatically stowed. If you have edited the subprogram proxy code in the code window, click **Stow <Module name>** from the **File** menu after generating.

Once generation has completed, the following two items exist:

- The generated subprogram proxy
- The application service definition generated into the Construct Spectrum Administration subsystem.

For more information about application service definitions, see **Application Service Definition Methods**, page 141.

Application Service Definition Methods

The subprogram proxy generates a method for each of the actions supported by an object subprogram. An application service definition generated for a maintenance object automatically includes the following methods:

- Delete
- Exists
- Get
- Initialize
- Next
- Store
- Update

For a browse subprogram:

- Browse

For any other type of subprogram:

- Default

If a subprogram proxy is generated using the same domain, business object, and version as another subprogram proxy, the new methods are also added to the application service definition. This allows a single application service definition to access both the maintenance and browse functions of a business object, for example.

Accessing the Maintain Application Service Definitions Panel

- To view application service definition records:
 - 1 Access the Construct Spectrum Administration subsystem main menu and enter “AA” in the Function field.
The Application Administration main menu is displayed.
 - 2 Enter “MM” in the Function field on the Application Administration main menu.
The Application Administration Maintenance menu is displayed.

Enter "AS" in the Function field on the Application Administration Maintenance menu.

The Maintain Application Service Definitions panel is displayed. From this panel you can create a method.

BSIF__MP Jan 30	Construct Spectrum Administration Subsystem Maintain Application Service Definitions	BSIF__11 3:15 PM
--------------------	---	---------------------

Action (A,B,C,D,M,N,P) _

Domain.....: DEMO_ *

Object.....: PRODUCT

Version.....: 01 / 01 / 01

Description.....: PRODUCT

Default subprogram proxy: PRODMSP_

Steplibs.....: _____ *

01	Method Name	Subprogram Proxy	Steplibs *
1	BROWSE	PRODBSP_	
2	DELETE		
3	EXISTS		
4	GET		
5	INITIALIZE		

Command: _____

Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---

confm help retrn quit flip pref bkwrđ frwrđ main

Appl Srvć Definition DEMO-PRODUC displayed successfully

Maintain Application Service Definitions Panel

Adding a Method

You can add custom methods to your maintenance or browse object. For example, if a maintenance object requires special processing that does not occur with one of the provided methods listed in the previous section, add a new method to implement the processing. Adding a new method involves three steps:

- 1 Creating the method
- 2 Updating the application service definition
- 3 Updating the library image file with the method

Optionally, you can perform modifications so that only the data that is required for your custom method is transmitted when the method is invoked. For more information about optimizing the handling of data for your custom method, see **Overriding Block Handling**, page 146.

Step 1 — Create the Method

- To create the method:
 - 1 Code the method in the USER-DEFINED-FUNCTIONS user exit for the subprogram and save your changes.
 - 2 If the subprogram does not currently include this user exit, regenerate it with the Object-Maint-Subp model and include the USER-DEFINED-FUNCTIONS user exit in your module.

For information about using the Object-Maint-Subp model and user exits, see **Object-Maint Models**, page 479, *Natural Construct Generation User's Manual*.

Step 2 — Update the Application Service Definition

- To update the application service definition:
 - 1 Type “M” in the Action field and type the domain, object, and version of the application service definition you are updating in the appropriate fields.
 - 2 Type the name of the method in the Method Name field.
Use the name that was specified when the method was created and added to the maintenance subprogram user exit.
 - 3 If the subprogram proxy for this business object's method is different from the default subprogram proxy specified for the business object, type the new subprogram proxy name in the Subprogram Proxy field; otherwise, leave the field blank.
 - 4 If the steplib for this business object's method is different from the default steplib specified for the domain, provide the new steplib name in the Steplibs field.
 - 5 Press Enter.
The method is added to the application service definition.

Step 3 — Update the Library Image File with the Method

The library image file resides on your client. The library image file must be updated with the valid methods for a particular business object. To update the LIF, simply download the subprogram proxy to the application's Visual Basic project. Construct Spectrum automatically updates the LIF with the new method.

- To update the library image file with the method:
- 1 In Visual Basic, open the project for your application and select **Download Generated Modules** from the **Construct Spectrum Add-In** menu.

For more information about using the **Download Generated Modules** command, see **Downloading the Generated Modules**, page 131 in *Developing Client/Server Applications*.
 - 2 Download the subprogram proxy definition to your project.
A maintenance subprogram proxy has the suffix "MSP" and a browse subprogram proxy has the suffix "BSP".
 - 3 Save your changes and run the project.
The new method is available for use in your application.

For web applications you need to regenerate the ABO, page handler and HTML template. For more information see, **Generating and Customizing Page Handlers**, page 85 and **Generating and Customizing HTML Templates**, page 105, *Developing Web Applications*.

Overriding the Domain's Steplib Chain

All business objects in an application service definition share the same domain. All business objects within a domain are accessed using the domain's steplib chain. You can, however, override the steplib chain for each business object or method defined in your application service definition.

- To override the domain's steplib chain:
- 1 Access the Maintain Application Service Definitions panel.

For information about accessing this panel, see **Accessing the Maintain Application Service Definitions Panel**, page 141.

BSIF_MPConstruct Spectrum Administration SubsystemBSIF_11
Jan 30Maintain Application Service Definitions3:15 PM

Action (A,B,C,D,M,N,P) _

Domain..... DEMO *
Object..... PRODUCT
Version..... 01 / 01 / 01
Description..... PRODUCT
Default subprogram proxy: PRODMSP_
Steplibs..... *

01	Method Name	Subprogram Proxy	Steplibs *
1	BROWSE	PRODBSP_	
2	DELETE		
3	EXISTS		
4	GET		
5	INITIALIZE		

Command:

Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
confm help retrn quit flip pref bkwrд frwrд main
Appl Srvс Definition DEMO-PRODUC displayed successfully

Maintain Application Service Definitions Panel

- 2 Display the application service definition you want to modify.
- 3 Type “M” in the Action field. For each method or business object that requires a special steplib, type the steplib name in the Steplibs field or press PF1 to select from a list of available steplibs.
- 4 Press Enter to update the application service definition.

Overriding Block Handling

The subprogram proxy optimizes level 1 parameter block handling for the default methods provided with your object maintenance and object browse subprograms. This optimization ensures that only the required data for a particular method is sent from the server to the client. This section describes the default block handling provided with the subprogram proxy and how to override this block handling, if necessary.

Default Block Handling

The following tables define which level 1 blocks are sent for each default method in your object maintenance and object browse subprograms.

Object Maintenance Subprogram Blocks Sent to the Server

Function	Business Object Data	Business Object Key	Restricted Data	CDAOBJ2 (function)	CDPDA-M (message)
DELETE		X	X	X	
EXISTS		X		X	
GET		X		X	
INITIALIZE				X	
NEXT		X		X	
STORE	X			X	
UPDATE	X		X	X	

Object Maintenance Subprogram Blocks Returned to the Client

Function and Flags	Business Object Data	Business Object Key	Restricted Data	CDAOBJ2 (function)	CDPDA-M (message)
DELETE and (Error = True or Return Object = False)				X	X
DELETE and (Clear After = False)			X	X	X
GET and Exists = False				X	X
EXISTS				X	X
NEXT and Exists = False				X	X
STORE and (Error = True or Return Object = False)				X	X
STORE and (Clear After = False and Derived Data = False)			X	X	X
UPDATE and (Error = True or Return Object = False)				X	X

Function and Flags	Business Object Data	Business Object Key	Restricted Data	CDAOBJ2 (function)	CDPDA-M (message)
UPDATE and (Clear After = False and Derived Data = False)			X	X	X
All Other Combinations	X		X	X	X

Browse Subprogram Blocks — Sent from the Client

Function	Key Data	Row Data	Restricted Data	CDBRPDA (function)	CDPDA-M (message)
BROWSE	X		X	X	

Browse Subprogram Blocks — Returned to the Client

Function	Key Data	Row Data	Restricted Data	CDBRPDA (function)	CDPDA-M (message)
BROWSE	X	X	X	X	X

Specifying Overrides

You can override the default block handling rules shown in the previous tables and provide your own rules. For example, if you add a new method, you can specify which blocks are to be sent to the client. By default, custom methods are set to transmit all data blocks.

Overriding block handling involves two steps:

- 1 Specifying block handling on the server
- 2 Specifying block handling on the client

In the first task, modify the subprogram proxy to specify which data blocks are sent from the server to the client. In the second task, modify the library image file to specify which data blocks are sent from the client to the server.

Step 1 — Specify Block Handling On the Server

Every level 1 data block in a subprogram's parameter data can be set up with block handling overrides. Hand-code these overrides in the subprogram proxy's SET-RETURN-BLOCKS user exit.

If you need to include this user exit in your subprogram proxy, regenerate the proxy. For information about regenerating the subprogram proxy, see **Generating a Subprogram Proxy**, page 132.

This section includes two procedures: one to unconditionally disable a block so that it is never sent to the client and another to conditionally disable a block so that it is sent to the client only if certain conditions are met.

- To disable a block unconditionally so that it is never sent to the client:

Note: You must code the statements in this user exit as part of a DECIDE FOR statement.

- 1 In the SET-RETURN-BLOCKS user exit of the subprogram proxy, reset any block indicator that is not to be sent to the client.
Block indicators identify a data block and are named #PDA.#RB-*blockname*, where *blockname* is the name of the level 1 variable that defines the block.
- 2 Add the following code:

```
WHEN #SPC-TRUE
RESET #PDA.#RB-BLOCKNAME/* Unconditional assignment
```

- To conditionally send blocks to the client:

In the SET-RETURN-BLOCKS user exit of the subprogram proxy, add a DECIDE clause that resets certain block selectors based on a condition. For example:

```
**SAG DEFINE EXIT SET-RETURN-BLOCKS
/* Do not return restricted data on a delete
WHEN CDAOBJ2.#FUNCTION = CDLMETHOD.DELETE
    RESET #PDA.#RB-CUSTMSR
/* Do not return object or restricted data on existence check
WHEN CDAOBJ2.#FUNCTION = CDLMETHOD.EXISTS
    RESET #PDA.#RB-CUSTMSA
    #PDA.#RB-CUSTMSR
**SAG END-EXIT
```

Adhere to the following guidelines when assigning the blocks:

- Know the name of each block that you are assigning.
The block name format is #PDA.#RB-*blockname* where *blockname* is the name of the level 1 field.
- Reset only those blocks that are not to be returned to the client.

Specify Block Handling On the Client

For information about specifying block handling on the client, see **Step 3 — Update the Library Image File with the Method**, page 144.

Versioning

New versions of a subprogram proxy can be created without affecting older versions. The version number that you specify when entering the model input parameters is part of the key used to store the associated application service definition. Versioning allows maintenance of a system to take place without affecting existing applications. Each request issued from the client includes its required version number.

Tip: When creating a new version of a subprogram proxy, use a new name for the subprogram proxy. Otherwise, the existing version will be overwritten.

Security Implications

Security definitions do not include the version number. This means that if the only thing about the subprogram proxy that changes is the version number, it will automatically be included in existing security definitions for the domain and business object name specified. If it requires a new security definition, the subprogram proxy domain or business object name should be changed to force the creation of a new application service definition. This new application service definition can then be secured as necessary.

Support for Debugging

Subprogram proxies automatically support the DATASIZES and INITIALIZE trace options which are useful when debugging an application. These options return the size of the data blocks and their initialized values. You can add additional trace options in the subprogram proxy's USER-TRACE-COMMANDS user exit.

For more information about debugging, see **Debugging Your Client/Server Application**, page 201.

USING BUSINESS DATA TYPES

This chapter describes business data types (BDTs) as they relate to client/server and web applications. It gives you a good understanding of the composition of BDTs and how to create and use them.

The following topics are covered:

- **Overview**, page 154
- **Understanding and Using BDTs**, page 155
- **Creating and Customizing BDTs**, page 176

Overview

The first section of this chapter is of particular interest to users of BDTs. It discusses the concept of BDTs in general terms and gives you a good understanding of the benefits of using BDTs and how they work. The second section is of interest to authors of BDTs. It discusses how to create and customize BDTs in both the client/server and web framework.

BDTs provide a way to present data to the user in a format that is consistent and based on business conventions rather than on programming language conventions. For example, a BDT could format a phone number with dashes (-) or some other delimiter value so that it is easily recognized by the user as a phone number.

To accomplish this, BDTs convert data values between simple internal Visual Basic data types (such as String, Long, Currency, Date, and Boolean) and values that are displayed to the user in a browse or maintenance dialog.

Construct Spectrum also uses BDTs to create sample strings to calculate the length of GUI controls.

Understanding and Using BDTs

There is some commonality between BDTs that are used in client/server applications and web applications. The following section discusses BDTs as they relate to both the client/server and web framework.

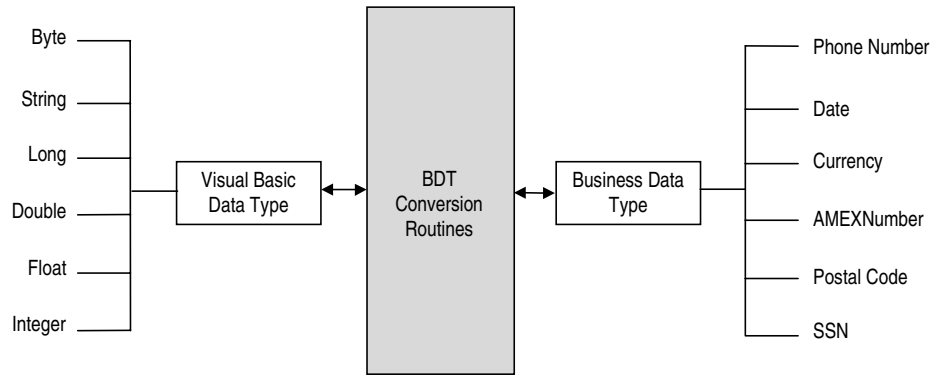
Benefits of BDTs

Using business data types offers three primary benefits:

- consistency
BDTs ensure that a specific data type is displayed in the same format throughout the application.
- flexibility
BDTs recognize a variety of input formats which makes using the application easier.
- accuracy
BDTs centralize the validation code for a data type and provide a consistent mechanism for returning validation error messages.

Relationship With Visual Basic Data Types

The relationship between Visual Basic data types and business data types is many-to-many. That is, a Visual Basic Double variable can represent more than one BDT, such as Phone Number, AMEX Number, or Currency. Conversely, a Phone Number BDT could be mapped to Visual Basic String, Double, or Float variables. The Visual Basic data types to which a BDT can be applied depend on the considerations written into the BDT's conversion routine.



Relationship Between Visual Basic Data Types And Business Data Types

Construct Spectrum includes a set of standard BDTs. You can use these BDTs as they are or you can customize them. You can also write your own BDTs. The client frameworks use an open architecture so that you can add business data types tailored to your application specifications. If you find there is a piece of information whose format you are constantly validating, consider writing a BDT to handle it. Once a BDT has been created, you can use it in other applications.

Composition of BDTs

A BDT is composed of a name, a conversion routine, and the list of modifiers it can use.

Name

Applications need only the name of the appropriate BDT to perform the conversion to and from a displayable value.

Conversion Routine

The conversion routine converts data between an internal Visual Basic data type and a displayable format.

The `BDTConversion` object is used internally by BDT conversion routines. When the application calls one of the BDT controller's conversion methods, the controller creates a `BDTConversion` object and initializes it with details about the conversion requested. For example, the BDT controller will supply the BDT name, any modifiers associated with it and any Natural format provided. The BDT controller then calls the conversion routine for the specified BDT, passing the `BDTConversion` object as a parameter.

The conversion routine uses the properties of the `BDTConversion` object to determine what type of conversion to perform (convert to display, convert from display, or create sample string), to get information about the modifiers used, the Natural format specified, and to return the converted value.

Modifiers

Use modifiers to override the default conversions that are performed by a BDT's conversion routine.

Using Business Data Types

Each time an application uses a business data type, it involves a number of elements. This section provides detail on what these elements are and how they relate to business data types.

The BDT Controller

The BDT controller knows about all the BDTs that the application uses. This is because the application registers all of its BDTs with the BDT controller when the application is started. Whenever an application uses a BDT, it relies on the BDT controller to locate and call the associated conversion routine. The process follows these steps:

- 1 The application calls the BDT controller and passes it all the necessary parameters, including the name of the BDT and the value to be converted.
- 2 The BDT controller locates the conversion routine for the BDT.
- 3 The BDT controller calls the conversion routine, forwarding the parameters from the application.

- 4 The conversion routine does the conversion and returns the result to the BDT controller.
- 5 The BDT controller returns the result to the application.

This process means that the application needs only the name of the BDT to accomplish the required conversion.

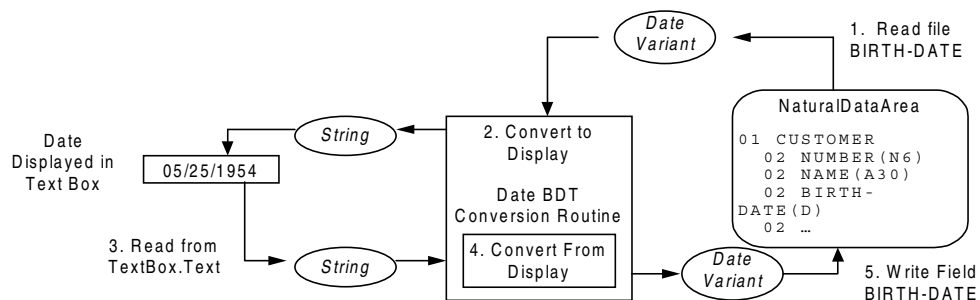
The BDT controller is declared in the client/server framework as follows:

```
Public BDT As New BDTController
```

In the web framework, the BDTController object is a global multi-use object, meaning that you can invoke properties and methods of this class as if they were global functions. You do not have to create an instance of this class first because one will automatically be created.

How the Client Frameworks Use BDTs

The client frameworks use BDTs to display values read from a NaturalDataArea on the client. The values are then displayed in GUI controls. The following diagram shows the process of reading a value from a NaturalDataArea on the client, displaying it in a GUI control where the user can modify the value, and copying the new value back to the NaturalDataArea on the client. Once the value is copied back to the data area on the client, it can be sent to the server. The BDT in the following diagram is named DATE and is applied to the field BIRTH-DATE:



Processing Date BDT Applied to BIRTH-DATE Field

- 1 Reading the field value from the `NaturalDataArea` on the client returns a Visual Basic Variant data type.
- 2 This value is formatted for display by the Date BDT's conversion routine through a call to `ConvertToDisplay`. The result is a String value.
- 3 The string is displayed on a form by assigning it to a GUI control. The value displayed in the text box can be edited by the user.
- 4 When the user is finished editing, the edited value is read from the `TextBox` control's `Text` property.
- 5 This string is converted back to a variant by the Date BDT's conversion routine through a call to `ConvertFromDisplay`. If the string does not contain a valid date or the conversion routine cannot interpret the user's value correctly, an error is returned.
- 6 The new value is assigned back to the field in the `NaturalDataArea` object, which can then be sent to the server, for example, in the case of an update to the server database.

In a web application, BDTs are implemented internally. You can change the BDT used by a field by coding a user exit. For more information, see **Customizing Page Handlers**, page 97, *Developing Web Applications*.

Calling Conversion Routines

When an application uses a BDT, the BDT controller calls the conversion routine. The conversion routine offers three services that affect the appearance of data:

- converting the value in a Visual Basic data type to display in business format
- converting the value from its business format display to a Visual Basic data type
- creating a sample display value that is representative of the display values produced by the BDT

In applying the second service, the conversion routine returns an error message if an inappropriate value is passed to it.

Convert to Display

The BDT controller's method, `ConvertToDisplay`, converts a value from a Visual Basic data type to a display format. This method takes the value, and either the name of a BDT or a Natural format, and returns a string that is formatted for display.

Its syntax is:

```
Function ConvertToDisplay(RawData As Variant, _
                        Optional BDTName As String, _
                        Optional NatFormatLength As String _
                        ) As String
```

For example, in a client/server application:

```
txtBirthDate.Text = BDT.ConvertToDisplay(custpda("BIRTH-DATE"), _
                                         "Date")
```

You can specify a BDT name, a Natural format such as N6.2, or both. If you do not specify a BDT name, the BDT controller uses the Natural format to choose an appropriate BDT first, and then calls that BDT's conversion routine.

If you do specify a BDT name, that BDT's conversion routine can use the optional Natural format to further refine how it performs the conversion or interprets the data. For example, the Numeric BDT uses the Natural format to determine how many decimal places to display. The Date BDT uses the Natural format as follows:

- If the Natural format is D, interpret the variant data as a date.
- If the Natural format is N6, P6, or A6, interpret the variant data as a date in the format YYMMDD.
- If the Natural format is N8, P8, or A8, interpret the variant data as a date in the format YYYYMMDD.

Convert from Display

The BDT controller's method `ConvertFromDisplay` converts a value from a display format to a Visual Basic data type. This method takes the display value, and either the name of a BDT or a Natural format, and returns a variant value that can be manipulated further by the application.

Its syntax is:

```
Function ConvertFromDisplay(FormattedData As String, _
    Optional BDTName As String, _
    Optional NatFormatLength As String _
) As Variant
```

For example, in a client/server application:

```
custpda("BIRTH-DATE") = BDT.ConvertFromDisplay(txtBirthDate.Text, _
    "Date")
```

You can specify a BDT name, a Natural format, or both. Using these optional parameters has the same result as in ConvertToDisplay.

Converting In-Place

The BDT controller's method ConvertInPlace allows you to validate and reformat a value in a GUI control such as in a LostFocus event. This method takes a formatted value by reference, internally calls ConvertFromDisplay, and then passes the result back to ConvertToDisplay which returns the new formatted result.

For example, in a client/server application:

```
Private Sub txtBirthDate_LostFocus()

    Dim stemp As String

    stemp = txtBirthDate.Text
    BDT.ConvertInPlace stemp, "Date"
    txtBirthDate.Text = stemp

End Sub
```

When the user moves out of the field, the field value is validated and, if valid, is reformatted according to the date format used by the BDT. For example, a user might enter "feb 5" and the input will be reformatted to the date format chosen, such as 2/5/1999, when the user leaves the field.

The syntax is:

```
Function ConvertInPlace(ByRef FormattedData As String, _
    Optional BDTName As String, _
    Optional NatFormatLength As String _
) As Variant
```

You can specify a BDT name, a Natural format, or both. Using these optional parameters has the same result as in `ConvertToDisplay`.

The return value is the value returned by the internal call to `ConvertFromDisplay`, so you can perform additional processing on the value entered by the user.

Creating Sample Values

The BDT controller has a method called `CreateSampleString` to create a sample displayable value for each BDT. This sample value can be used as a template to determine the dimensions of the associated control or to determine how wide a column in a browse dialog must be to display the BDT value.

Its syntax is:

```
Function CreateSampleString(Optional BDTName As String, _
    Optional NatFormatLength As String _
) As String
```

You can specify a BDT name, a Natural format, or both. Using these optional parameters allows you to further refine how the BDT performs the conversion or interprets the data.

The returned value can be used to calculate the required width of a `ListView` control column or a `TextBox` control used to display this business data type.

Using Modifiers

The processing to be performed by a business data type can be refined by means of special modifiers. Each business data type defines its own set of modifiers to provide the flexibility it needs.

Individual modifiers are separated by commas, and each modifier must be introduced by a name. Modifiers have names such as “TRIM”, “CASE”, “DEC”, and “ROUND.”

In calls to the conversion routines, use the format `name=value`, where `name` is the modifier you want to use and `value` controls the behavior of the conversion routine for the given modifier. Append modifiers to the BDT name parameter with commas. The following code invokes the Numeric BDT's conversion routine and uses the "DEC" modifier to specify that two decimal places should be displayed in the value and the "ZERO" modifier to suppress display of the value when it is 0.

For example, in a client/server application:

```
txtHours.Text = BDT.ConvertToDisplay(dblHours, _  
                                     "Numeric,DEC=2,ZERO=OFF")
```

For more information about the modifiers supported by each BDT, see **BDTs Supplied With Construct Spectrum**, page 166.

BDTs and Natural Formats

When you omit the BDT name in calls to `ConvertToDisplay`, `ConvertFromDisplay`, `ConvertInPlace`, and `CreateSampleString`, you must provide the Natural format. The BDT controller uses the Natural format to choose a BDT to use for the conversion.

It does this by calling a Natural-to-BDT mapper function. This function provides the most appropriate BDT to use for each Natural format.

The mapper function must be registered with the BDT controller. In the client/server framework, the mapper is implemented as a method of the `StandardBDTs` class and is registered in its `SelfRegister` method. For more information, see **Registering Your BDT**, page 181. For more information about registering BDTs in the web environment, see **Registering BDTs**, page 189.

Handling Errors Returned from a BDT Conversion Routine

The BDT controller has four properties that return error information from the conversion routines. These properties can be examined after a call to `ConvertFromDisplay` or `ConvertInPlace`. The BDT conversion routines place information in these properties if a conversion error occurs. The application can then examine the properties on return from the call and display the error to the user:

Property	Contents
<code>ErrorCode</code>	Contains the numeric error code. Each BDT can define its own error codes. The application can make program flow decisions based on this value.
<code>ErrorMsg</code>	Contains the error message. The message displayed should provide the user with useful information.
<code>ErrorPos</code>	Indicates the position of the first invalid character.
<code>ErrorLen</code>	Indicates the number of invalid characters.

An application can use the `ErrorPos` and the `ErrorLen` properties to set the `SelStart` and `SelLength` properties of a `TextBox` control to show the user where the invalid characters are.

Example code using error information properties in a client/server application:

```
Private Sub txtBirthDate_LostFocus ()

    Dim sdate As String

    sdate = txtBirthDate.Text
    BDT.ConvertInPlace sdate, "Date"
    If BDT.ErrorCode Then
        txtBirthDate.SelStart = BDT.ErrorPos
        txtBirthDate.SelLength = BDT.ErrorLen
        MsgBox BDT.ErrorMessage, vbExclamation
        txtBirthDate.SetFocus
    Else
        txtBirthDate.Text = sdate
    End If

End Sub
```

Warning: A conversion routine might set ErrorPos but not ErrorLen. In the sample code above, it will not cause problems.

How Spectrum Web Applications Use BDTs

Spectrum web applications use BDTs as a way to format and validate user input for display on web pages. No formatting or validation is done in the web browser, instead the work is done on the web server inside of the Spectrum web application component ABOInterface. To determine what BDTs to use, the page handler queries the ABO at runtime for the logical format each property provides. These logical formats are translated into BDT names. You can override the translation and logical formats of properties in a user exit called BDT.Overrides.

For example, in a specific page handler:

```

Private Sub ICSTPageHandler_Initialize(...)

    With m_ABOInterface
        Set .ABOObject = m_ABO

        .Init ERR_SESSION_KEY, m_RequestData.Session, m_RequestData.Request
        '<cst:EXIT Name="BDT.Overrides">
        ' For the CustomerPhoneNumber property use a phone BDT.
        .BDT("CustomerPhoneNumber") = "Phone"
        ' Use an alpha BDT for the logical format PostalCode.
        .LogicalFormatBDT("PostalCode") = "Alpha"
        '</cst:EXIT>
    End With
End Sub

```

When a maintenance or browse HTML template is parsed and FIELD tags are detected, the value to be displayed is formatted using the correct BDT.

When a user submits a web page that includes properties on an HTML form to be updated, the ABOInterface component is used to update these properties. Part of the process includes validating the data included on the form before updating the property. If a BDT validation error occurs, the property is flagged for an error and the user's action is cancelled. When the web page is returned to the user, the properties in error are highlighted in red (Internet Explorer) or an error graphic is displayed next to the field (Netscape) and the error messages are displayed at the bottom of the page.

For more information, see **Customizing Page Handlers**, page 97, *Developing Web Applications*.

BDTs Supplied With Construct Spectrum

This section describes the standard BDTs that are shipped with Construct Spectrum. Each of the following sections describes one BDT, lists the modifiers it supports, and describes what each modifier does.

Alpha

Apply the BDT Alpha to alphanumeric data.

Modifier	Description
TRIM=L T LT	Trims leading spaces (L), trailing spaces (T), or leading and trailing spaces (LT). Default is no trimming of spaces. This affects ConvertToDisplay and ConvertFromDisplay behavior.
CASE=U L	Forces the text into uppercase (U) or lowercase (L). Default is to not change the case. This affects ConvertToDisplay and ConvertFromDisplay behavior.

Boolean

Apply BDT Boolean to data that can have a value of either False or True.

Modifier	Description
EM=<False> <True>	Displays the <False> string for False and the <True> string for True. Default is EM=False True. ConvertFromDisplay compares the formatted data to the <False> and <True> strings and recognizes a match if the value matches unambiguously to the beginning of either string. This is not case-sensitive.

The following are examples of various types of edit mask values, user input, and each result.

<u>EM Value</u>	<u>Formatted Value</u>	<u>Raw Value</u>
EM=False True	T	True
	t	True
	tr	True
	TRU	True
	F	False
	false	False
	yes	Error: Invalid
	<blank>	Error: Invalid

Modifier	Description (continued)		
EM=True False	true	False	
	F	True	
EM=Off On	off	False	
	on	True	
	o	Error: Ambiguous	
EM= X	x	True	
	<blank>	False	
	xx	Error: Invalid	

Time

Apply the BDT Time to any time value. The Time BDT supports the following Natural formats.

Natural Format	Visual Basic Data Type	Description
T	Date, Variant	If the value is Null, ConvertToDisplay returns an empty string.
N7 or P7	Long, Single, Double, or Currency	The numeric value is interpreted as HHMMSSST.
A7	String	The alpha value is interpreted as HHMMSSST.

Numeric

Apply the BDT Numeric to any numeric data.

Modifier	Description
DEC= <i>n</i>	Forces the display of <i>n</i> decimal places. Default is to display as many decimal places as there are significant decimal digits when the Natural format is not provided, or to use a fixed number of decimal places if the Natural format is provided. In this latter case, use DEC=-1 to ignore the Natural format and display significant decimal digits only.
ROUND= <i>n</i>	Rounds the value to <i>n</i> decimal places. If <i>n</i> is negative, it rounds to the left of the decimal place. Default is no rounding.
GS=OFF ON	Used to suppress (OFF) or display (ON) group separators (thousands separators). Default is GS=OFF.
ZERO=OFF ON	Used to suppress (OFF) or display (ON) zero values. Default is ZERO=OFF.
SIGN=OFF ON	Used to suppress (OFF) or display (ON) the sign for positive numbers. Default is SIGN=OFF.
MULT= <i>n</i>	ConvertToDisplay multiplies the raw value by <i>n</i> . ConvertFromDisplay divides the value by <i>n</i> before returning the raw value. <i>n</i> can be any positive or negative numeric value except zero. Default is MULT=1.
SCIENTIFIC=OFF ON	Displays the value in normal (OFF) or scientific notation (ON). Default is SCIENTIFIC=OFF.

Modifier	Description (continued)
EM=xxx	xxx is any format string understood by the Visual Basic Format function. ConvertToDisplay uses the Format function to format the value according to that format string.
STRICT=OFF ON	Has no effect on ConvertToDisplay. Used by ConvertFromDisplay to determine how to deal with non-numeric characters in the formatted value. OFF quietly discards non-numeric characters and ON generates an error if the value contains non-numeric characters. Default is STRICT=ON

Currency

Apply the BDT Currency to any currency values.

Modifier	Description
ZERO=OFF ON	Used to suppress (OFF) or display (ON) zero values. Default is ZERO=ON.

Date

Apply the BDT Date to any date value. The Date BDT supports the following Natural formats.

Natural Format	Visual Basic Data Type	Description
D and T	Date, Variant	If the value is Null, ConvertToDisplay returns an empty string.
N6 or P6	Long, Single, Double, or Currency	The numeric value is interpreted as YYMMDD.

Natural Format	Visual Basic Data Type	Description (continued)
N8 or P8	Long, Single, Double, or Currency	The numeric value is interpreted as YYYYMMDD.
A6	String	The alpha value is interpreted as YYMMDD.
A8	String	The alpha value is interpreted as YYYYMMDD.

Referencing BDTs in Predict

You can attach a BDT name to a field in Predict by adding a keyword with the same name as the BDT and prefix it with 'BDT_'. For example, to cause an N8 field to be treated as a date value when displayed on a browse or maintenance dialog, add the keyword BDT_DATE to the field.

The following table displays the BDT keywords that are loaded into Predict during the installation process:

BDT	Predict Keyword
Alpha	BDT_ALPHA
Boolean	BDT_BOOLEAN
Currency	BDT_CURRENCY
Date	BDT_DATE
Numeric	BDT_NUMERIC
Phone	BDT_PHONE
PostalCode	BDT_POSTALCODE

BDT	Predict Keyword (continued)
-----	-----------------------------

Time	BDT_TIME
ZipCode	BDT_ZIPCODE

Defining BDTs

One of the most powerful things about BDTs is that you can customize existing BDTs or write your own. If there is a piece of information whose format you are constantly validating, consider writing a BDT to handle it. A perfect case for a customized BDT might be an organization-specific account number.

To define a BDT, you must provide a name, the list of modifiers it will support, the display format it will use, the Natural formats, and the variant data types it will support.

Name

A BDT name can be any consecutive string of characters, but it cannot include commas. Leading and trailing spaces are ignored, and uppercase and lowercase are considered identical.

Tip: To maintain consistency, follow the naming convention used in the Construct Spectrum client frameworks: use short names consisting of one or two words, and capitalize the first letter of each word.

Modifiers

Individual modifiers are separated by commas, and each modifier is introduced by a name.

Modifiers have names such as TRIM, CASE, DEC, and ROUND. Modifier names can be any consecutive string of characters but cannot include commas or the equal sign. Leading and trailing spaces are ignored, and uppercase and lowercase are considered identical. To maintain consistency, use the naming convention estab-

lished for the Construct Spectrum client frameworks: use short names consisting of a sequence of uppercase letters, digits, or a combination of uppercase letters and digits.

Supported Natural Formats

In addition to modifiers, all BDT handlers can be passed the format and length of the Natural variable that is to receive the contents of the converted strings. The BDT handlers can use this information to apply, for example, truncation rules, or insert defaults.

When you omit the BDT name in calls to `ConvertToDisplay`, `ConvertFromDisplay`, `ConvertInPlace`, and `CreateSampleString`, you must provide the Natural format. The BDT controller uses the Natural format to choose a BDT to use for the conversion.

It does this by calling a Natural-to-BDT mapper function that is included in the Construct Spectrum client frameworks. The mapper function must also be registered with the BDT controller.

Returning An Appropriate Variant Type

When converting from formatted data to raw data, decide what type of variant to use for the raw data. If you have a phone number BDT, you can return the phone number as, for example, a Visual Basic Double, String, Currency, or an array. All of these data types have enough precision to store all digits of a phone number. Choose a data type that is convenient for the application programmer.

The data type you return may also depend on the Natural format passed into the conversion routine. For example, a seven-digit telephone number with area code could be stored in an A10 field or in an N10 or P10 field. The conversion routine could return a String variant if the Natural format is A, and a Double variant if the Natural format is N or P.

Returning Conversion Error Information

Conversion routines return conversion error information to the BDT controller in the error properties of the BDTConversion object. The BDT controller copies these properties to its own properties having the same names. The client application then examines the error properties of the BDT controller to determine if an error occurred.

When returning error information, the most important property to set in the conversion routine is `ErrorCode`. If this property is not set, the BDT controller and the client application do not know that an error has occurred because they make program flow decisions based on `ErrorCode`.

If you set `ErrorCode`, you should also set `ErrorMsg`, giving the client application a message to display to the user.

To provide the most information to the client application, set `ErrorPos` and then optionally set `ErrorLen`.

When converting from formatted data to raw data, your conversion routine can range from very forgiving in the input allowed to very strict. For example, a very forgiving conversion routine might throw away any non-numeric characters in a numeric BDT without returning an error, while a very strict conversion routine might require the input to match a rigid format to be converted without error.

A forgiving conversion routine will be easier to code because it will contain comparatively few validations. Coding a strict conversion takes more time and may be more difficult to code if the routine must examine the input character-by-character to determine if it is valid. However, a strict conversion routine means your error messages can be more informative.

Runtime Error Handling

Your conversion routine should also ensure that it uses Visual Basic runtime error handling to trap any runtime errors that might occur. If they are not trapped by the conversion routine, Visual Basic transfers the error up the call chain to the first enabled error handler. The BDT controller that called the conversion routine does have an enabled error handler; and converts the error into the number &H80040206 - An unhandled run-time error occurred when calling the method %1 in object %2:Error %3, %4.

The client application is typically not prepared to handle a runtime error that occurs in a conversion routine that it called indirectly.

Therefore, it is imperative that Visual Basic runtime errors are trapped in the conversion routines and translated into BDT-specific errors that are documented and returned in the error properties of `BDTConversion`.

Creating and Customizing BDTs

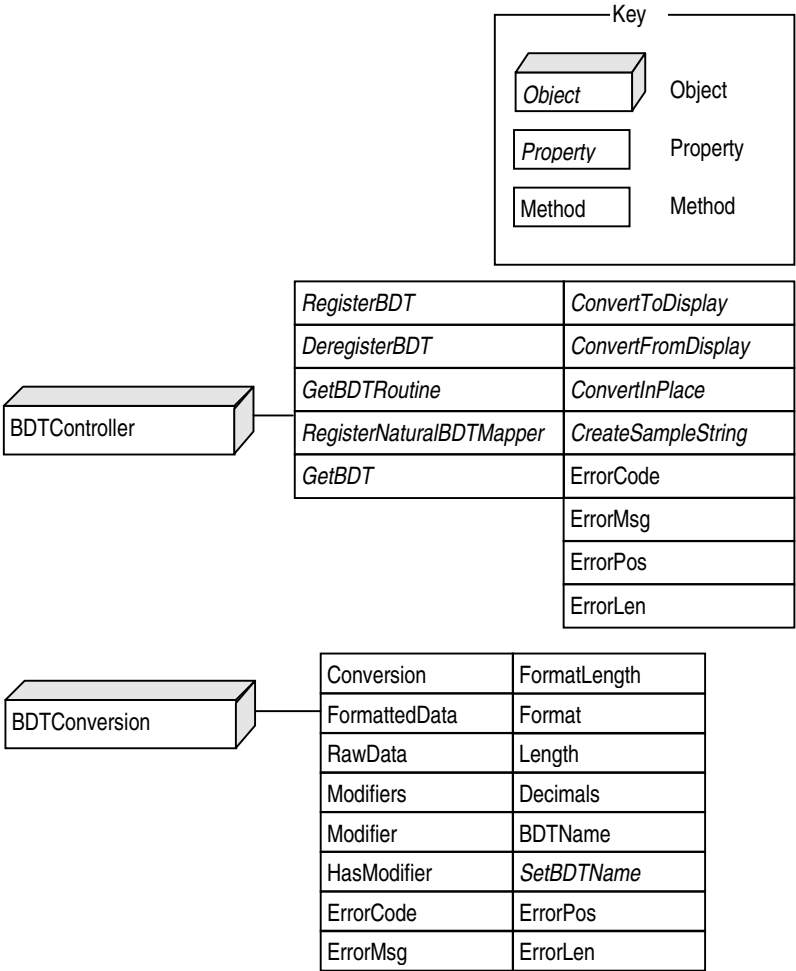
This section discusses how to create and customize BDTs. The client/server framework and web framework use an open architecture so that you can add business data types tailored to your application specifications.

BDTs and the Client/Server Framework

This section discusses how the client/server framework uses BDTs. For information about creating BDTs for the web framework, see **BDTs and the Web Framework**, page 187.

Understanding the BDT Objects

The Construct Spectrum client framework has two objects that support BDTs: BDTController and BDTConversion. The properties and methods of these objects are shown in the following diagram:



Properties and Methods of the BDT Objects

The BDTController object is used by the application to register its BDTs (the startup code in the Construct Spectrum client framework does this for you) and to call BDT conversion routines.

Each of the methods `ConvertToDisplay`, `ConvertFromDisplay`, `ConvertInPlace`, and `CreateSampleString`, and the error properties `ErrorCode`, `ErrorMsg`, `ErrorPos`, and `ErrorLen` are discussed in separate sections in this chapter. The remaining methods are related to registering BDTs, which is covered in **Registering Your BDT**, page 181.

Creating BDT Conversion Routines

BDT conversion routines must be implemented as public methods of an OLE automation object. This object could reside in an in-process server, an out-of-process server, or as a class in the Visual Basic project.

All BDT conversion routines must have the following syntax:

```
Public Sub xxx(BDTC As BDTConversion)
```

where *xxx* can be any name.

The following table describes all of the properties and methods of the `BDTConversion` object for the client/server framework. In the examples in this table, assume the following call has been made:

```
strHours = BDT.ConvertToDisplay(dblHours, _
                                "Numeric,ZERO=OFF,ROUND=2,STRICT=ON", _
                                "N3.2")
```

Property or Method	Description
Conversion	<p>Tells the conversion routine what type of conversion to perform. Can be one of the following constants:</p> <p><code>bdtConvertToDisplay</code> <code>bdtConvertFromDisplay</code> <code>bdtCreateSampleString</code></p>
FormattedData and RawData	<p>When <code>Conversion = bdtConvertFromDisplay</code>, the conversion routine should read the value in <code>FormattedData</code>, convert it into a Visual Basic data type, and assign the new value to <code>RawData</code>.</p>

Property or Method Description (continued)

When Conversion = bdtConvertToDisplay, the conversion routine should read the value in RawData, format it for display, and assign the formatted string value to FormattedData.

When Conversion = bdtCreateSampleString, the conversion routine should assign a sample string to FormattedData.
For example:

```
With BDTC
  Select Case .Conversion
    Case bdtConvertFromDisplay
      .RawData = cvtToRaw(.FormattedData)
    Case bdtConvertToDisplay
      .FormattedData = cvtToDisp(.RawData)
    Case bdtCreateSampleString
      .FormattedData = createSample()
  End Select
End With
```

Modifiers Returns the number of modifiers specified by the caller.
With the example above, Modifiers would return 3.

Modifier Returns the value of a specific modifier or can be used to enumerate the modifiers used. Using the example above:

```
With BDTC
  Print .Modifier("ZERO") ' Prints "OFF"
  Print .Modifier(1)      ' Prints "ZERO"
  Print .Modifier(2)      ' Prints "ROUND"
  Print .Modifier(.Modifier(1)) ' Prints "OFF"
End With
```

Note: Modifier names, such as ZERO or ROUND, are passed to the BDT conversion routine in uppercase. You do not have to use case-sensitive string comparisons when checking which modifiers have been used.

Property or Method Description (continued)

HasModifier	<p>Returns True if a given modifier has been used, and False if not. Using the example above:</p> <pre> With BDT Print .HasModifier("ZERO") ' Prints True Print .HasModifier("ROUND") ' Prints True Print .HasModifier("DEC") ' Prints False Print .HasModifier("#\$%^&") ' Prints False End With </pre>
FormatLength	Returns the Natural format string used in the call. Using the example above, FormatLength would return "N3.2".
Format, Length, and Decimals	Returns the format letter, length, and decimal portions, respectively, of the Natural format string used in the call. Using the example above, Format contains "N", Length contains 3, and Decimals contains 2.
BDTName	<p>Returns the name of the BDT from the call. Using the example above, BDTName contains "Numeric".</p> <p><i>Note: BDT names are passed to the BDT conversion routine with the capitalization used when the BDT was registered. For example, if RegisterBDT was called to register the BDT "mIxEdCase," and then ConvertToDisplay was called for the BDT "MixedCase," BDTC.BDTName would contain "mIxEdCase."</i></p>
SetBDTString	This method can be used to change the BDT name and the modifiers in the BDTConversion object.
ErrorCode, ErrorMsg, ErrorPos, and ErrorLen	The conversion routine should assign values to these properties if a conversion error occurred.

You can examine the conversion routines in the StandardBDTs.cls and CustomBDTs.cls modules to see how these properties and methods are used in BDTs.

Registering Your BDT

To make a BDT available for the application to use, the BDT controller needs to know about the BDT. This is done by registering the BDT with the BDT controller.

To register the BDT, tell the BDT controller the name of the BDT and provide a pointer to the BDT conversion routine. Conversion routines must be implemented as methods of an OLE automation object. To invoke a method, you must have a reference to the object in an object variable. Thus, the pointer to the BDT conversion routine consists of a reference to an object and the name of a public method in that object.

The registration process is shown in this example from the Construct Spectrum client framework. This code creates the BDT controller and instantiates the objects that contain the BDT conversion routines:

```
Public BDT As New BDTController

Private Sub InitializeBDTs()

    Dim StandardBDTs As New StandardBDTs
    Dim CustomBDTs As New CustomBDTs

    StandardBDTs.SelfRegister BDT
    CustomBDTs.SelfRegister BDT

End Sub
```

The registration actually occurs in the SelfRegister methods. The following example shows registration within the StandardBDTs class:

```
Public Sub SelfRegister(BDT As BDTController)
    BDT.RegisterBDT "Alpha", Me, "Convert_Alpha"
    BDT.RegisterBDT "Boolean", Me, "Convert_Boolean"
    BDT.RegisterBDT "Numeric", Me, "Convert_Numeric"
    BDT.RegisterBDT "Currency", Me, "Convert_Currency"
    BDT.RegisterBDT "DateTime", Me, "Convert_DateTime"
End Sub
```

In the RegisterBDT method, the first parameter is the name of the BDT, the second parameter is the object reference, and the third parameter is the name of a conversion routine in the object (a public method).

The BDT controller maintains a list of all BDT names internally along with the object reference and method to call for each.

To deregister one or more BDTs, call the `DeregisterBDT` method as follows:

```
BDT.DeregisterBDT           ' Deregisters all BDTs.
BDT.DeregisterBDT Me       ' Deregisters only the BDTs in the
                           ' specified object.
BDT.DeregisterBDT Me, "Numeric" ' Deregisters only the Numeric BDT in
                           ' the specified object.
```

Deregistering BDTs is useful if you need to release all references to an object so that the object can be destroyed. You can then recreate the object and re-register all BDTs it implements.

If you need to locate the conversion routine for a given BDT, you can use the `GetBDTRoutine` to return the object reference and method name of the conversion routine. Its syntax is:

```
Sub GetBDTRoutine(BDTName As String, _
                  ByRef Handler As Object, _
                  ByRef ProcName As String)
```

If the BDT name has not been registered, `Handler` will contain `Nothing` and `ProcName` will contain an empty string on return.

Creating a Natural-to-BDT Mapper

A Natural to BDT mapper function is called by the BDT controller when the application uses a conversion function and, instead of providing the name of a BDT, provides the Natural format. The mapper provides the most appropriate BDT to use for each Natural format.

A mapper function must be registered with the BDT controller just as BDTs are registered. In Construct Spectrum, the mapper is implemented as a method of the StandardBDTs class and is registered in its SelfRegister method:

```
Public Sub SelfRegister(BDT As BDTController)
    ...
    BDT.RegisterNaturalBDTMapper Me, "NaturalBDTMapper"
End Sub

Public Function NaturalBDTMapper(Format As String, _
                                Length As Long, _
                                Decimals As Integer) As String

    Dim sbdtstring As String

    ' BDT name was not provided. Pick a default BDT name based on the
    ' Natural format.
    Select Case Format
    Case "A": sbdtstring = BDT_ALPHA & ",TRIM=LT"
    Case "B": sbdtstring = BDT_ALPHA
    Case "D": sbdtstring = BDT_DATE
    Case "F": sbdtstring = BDT_NUMERIC
    Case "I": sbdtstring = BDT_NUMERIC
    Case "L": sbdtstring = BDT_BOOLEAN
    Case "N": sbdtstring = BDT_NUMERIC
    Case "P": sbdtstring = BDT_NUMERIC
    Case "T": sbdtstring = BDT_TIME
    End Select

    NaturalBDTMapper = sbdtstring
End Function
```

The GetBDT method of the BDT controller returns the name of the BDT used for the given Natural format. Using the mapper listed above:

```
Print BDT.GetBDT("D")      ' Prints "Date".
Print BDT.GetBDT("L")      ' Prints "Boolean".
Print BDT.GetBDT("N6")     ' Prints "Numeric".
Print BDT.GetBDT("N6.2")   ' Prints "Numeric".
```

Other Considerations

A list of other considerations for creating BDTs follows.

One Conversion Routine — Multiple BDTs

When you register a BDT, you can use the same function pointer for multiple BDTs, as the following example shows:

```
BDT.RegisterBDT "AccountNumber", Me, "Convert_Numbers"  
BDT.RegisterBDT "DeptNumber", Me, "Convert_Numbers"  
BDT.RegisterBDT "GroupNumber", Me, "Convert_Numbers"  
BDT.RegisterBDT "FileNumber", Me, "Convert_Numbers"
```

When the application uses the BDT, the conversion routine checks the BDT name to determine what conversion to perform:

```
Public Sub Convert_Numbers (BDTC As BDTConversion)  
    Select Case BDTC.BDTName  
        Case "AccountNumber"  
            ...  
        Case "DeptNumber", "GroupNumber"  
            ...  
        Case "FileNumber"  
            ...  
    End Select  
End Sub
```

Where To Place The Conversion Routine

When you create a new BDT conversion routine, you can add it to an existing class or you can create a new class.

Using the Construct Spectrum client framework, adding the conversion routine to the existing StandardBDTs or CustomBDTs module requires the fewest changes to the code. You need only add a method and then change the SelfRegister method to register the new BDT.

Warning:

When you save the updated version of the class, ensure that you do not overwrite the version in the Construct Spectrum client framework directory, unless you want the update to affect all existing and new projects that point to that class.

If you create a new class, change the InitializeBDTs procedure in the Startup module to instantiate the class and call its SelfRegister method.

BDT Overrides

When the same BDT name is registered with the BDT controller more than once, the last one registered is used. This feature can be used if you want to replace a supplied BDT (conversion routine) with your own. As long as you register your BDT conversion routine last, it will be called instead of the supplied one.

If you are replacing a supplied BDT routine with your own, you could use the GetBDTRoutine method to get the pointer to the current BDT routine before registering your own, and call the original BDT routine in certain cases:

```

Private m_OldHandler As Object
Private m_OldProcName As String

Public Sub SelfRegister(BDT As BDTController)

    ' Save the pointer to the old routine.
    BDT.GetBDTRoutine "Currency", m_OldHandler, m_OldProcName
    BDT.RegisterBDT "Currency", Me, "Convert_Currency"

End Sub

Public Sub Convert_Currency(BDTC As BDTConversion)

    With BDTC
        Select Case .Conversion
            Case bdtConvertToDisplay
                ' Custom conversion.
                ...
            Case Else
                ' Call the old routine.
                InvokeMethod m_OldHandler, m_OldProcName, Array(BDTC)
        End Select
    End With

End Sub

```

This example method uses the `InvokeMethod` procedure in the Construct Spectrum client framework. The `InvokeMethod` procedure can call any public method of any object by passing in a reference to the object and the name of the method in a string.

Referencing BDTs in Your Application

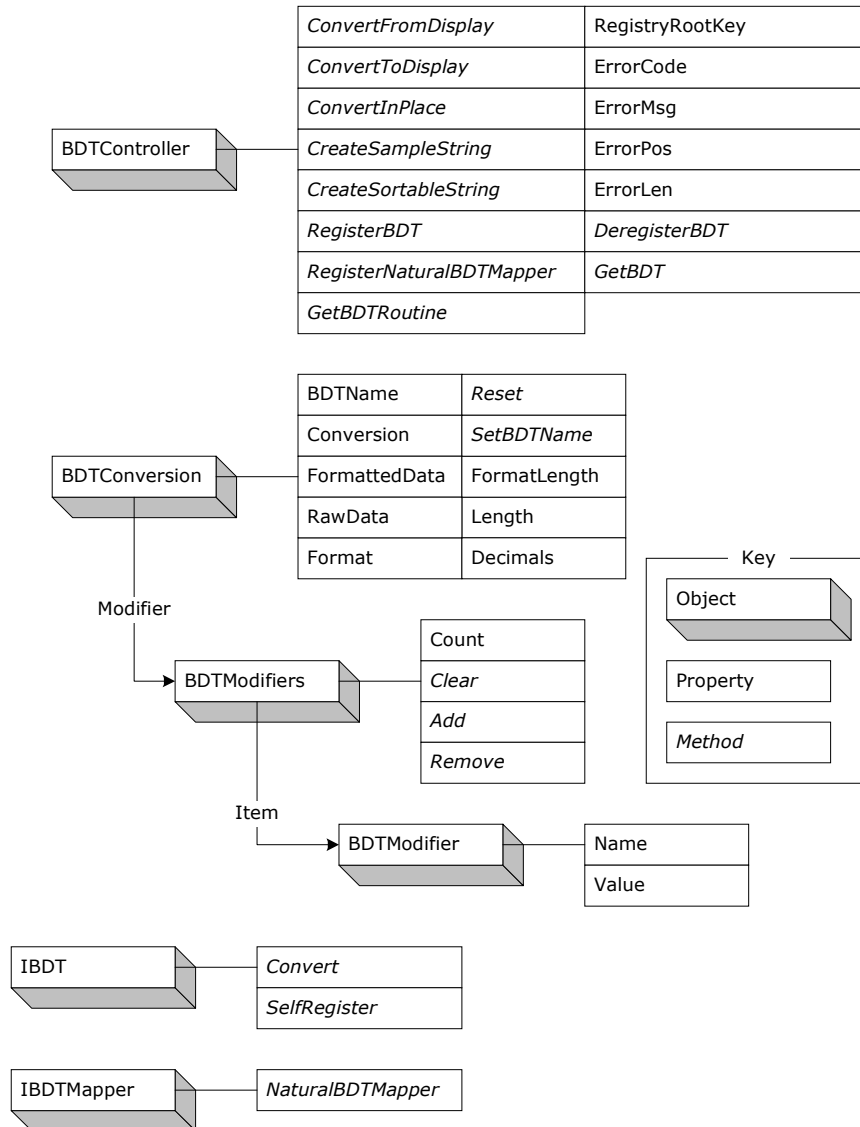
Each BDT in the `StandardBDTs` and `CustomBDTs` classes of the Construct Spectrum client framework has an associated named constant in `BDTSupport.bas`. The name of the constant is the same as the name of the BDT except that it is in uppercase and prefixed with “BDT_”. Instead of using the BDT name directly through the application, use the named constant. This allows the Visual Basic compiler to check that the BDT is defined in the framework. When you create your own BDTs, ensure that you add the named constants to `BDTSupport.bas`.

BDTs and the Web Framework

This section discusses BDTs and how they relate to the Web framework.

Understanding the BDT Objects

The Construct Spectrum web application framework uses objects in the BDTLib6 object library to support BDTs. These objects and their properties and methods are shown in the following diagram:



Implementing BDTs in the Web Framework

The following describes the steps needed to implement a BDT in a web framework.

What is a BDT Class

In the Construct Spectrum web framework, BDT conversion routines reside inside a Visual Basic class module that implements the IBDT interface. We use the name BDT class to refer to these types of Visual Basic classes. Therefore, a BDT class:

- Implements the IBDT interface.
- Contains a BDT conversion routine.

The IBDT interface has two methods:

Method	Description
Convert	The BDT controller calls this method to perform a conversion. It passes in a BDTConversion object that contains details about the conversion that is supposed to be performed.
SelfRegister	In this method, the BDT class must tell the BDT controller the names of the BDTs that it implements.

Registering BDTs

When an application needs to use a BDT, it simply calls the BDT controller and specifies the name of the BDT that it wants to use (such as “Boolean”). The BDT controller knows how to locate and call the BDT conversion routine by registering the BDT class with the BDT controller.

This allows the BDT controller to associate the name of a BDT with a BDT class. When the BDT controller needs to call the conversion routine, it creates an instance of the BDT class, gets a reference to the class’ IBDT interface, and finally calls the Convert method.

You can use two techniques to register BDT classes with the BDT controller:

- 1 Use the Windows Registry to register BDT classes
- 2 Explicitly register BDT classes

Use the Windows Registry to Register BDT Classes

The first technique for registering a BDT class is to use the Windows Registry to list all of the BDT classes installed on the PC. For the standard BDTs supplied with the web framework, the following excerpt from the Registry shows how this is done:

```
HKEY_LOCAL_MACHINE
    Software
        Software AG
            Business Data Types
                Alpha
                    ProgID = StandardBDTs6.BDTAlpha
                AlphaMultiline
                    ProgID = StandardBDTs6.BDTAlphaMultiline
                Boolean
                    ProgID = StandardBDTs6.BDTBoolean
                ...
```

Notice the names of the BDTs under the Business Data Types key. Each BDT key contains a ProgID string value that tells the BDT controller the programmatic ID (progID) of the class that implements the BDT conversion routine. Knowing the progID allows the BDT controller to create an instance of the BDT class.

The following example shows how the BDT controller locates and calls the BDT conversion routine for the Boolean BDT. The following steps illustrate the process:

- 1 The BDT controller looks up the ProgID value under the Boolean key and finds the name "StandardBDTs6.BDTBoolean".
- 2 It uses the Visual Basic CreateObject function to create an instance of this class. By using the facilities of COM, CreateObject loads the ActiveX DLL that implements the BDT class (StandardBDTs6.dll) and creates an instance of the BDT class (BDTBoolean).

- 3 The BDT controller calls the SelfRegister method in the IBDT interface implemented by the BDT class. The following example shows the SelfRegister method:

```
Private Sub IBDT_SelfRegister(BDT As BDTController)
    BDT.RegisterBDT "Boolean", Me
End Sub
```

The BDT class calls the BDT controller's RegisterBDT method, passing in the name of the BDT and an object reference to itself.

- 4 The RegisterBDT method in the BDT controller stores the BDT name and the object reference in an internal table. The BDT controller uses this table as a cache to store object references so it doesn't have to create a new instance of the BDT class each time the application uses the BDT.
- 5 The BDT controller now has a reference to an instance of the BDT class, and calls the Convert method in the class' IBDT interface.

The next time the application uses the Boolean BDT, the BDT controller looks in its internal table first and finds the BDT name and object reference. It can then call the Convert method immediately without having to perform any of the previous steps.

Placing BDT classes in an ActiveX DLL and using the Windows Registry to list them has the following advantages:

- BDT classes can be shared by many applications. By implementing BDT classes in an ActiveX DLL, they can be developed, tested, and enhanced separate from any application.
- An application that needs to use BDTs does not have to explicitly register all of the BDTs that it will use. The BDT controller simply locates and loads the BDT class containing the conversion routine dynamically at run-time whenever it is needed.
- BDTs can be added to a PC by simply installing the ActiveX DLL that contains them and then registering the new BDTs to the BDT controller by adding Registry keys under HKEY_LOCAL_MACHINE\Software\Software AG\Business Data Types.

Explicitly Register BDT Classes

The second technique for registering a BDT class is to explicitly call the BDT controller's RegisterBDT method in your application's startup code. For example, you may have code such as the following:

```
Public Sub Main

    ...

    ' Register the BDTs used by the application.
    RegisterBDT "Phone", New BDTPhone
    RegisterBDT "ZipCode", New BDTZipCode
    RegisterBDT "UPC", New BDTUPC

    ...

    ' Show the application's main form.
    frmMain.Show

End Sub
```

In this example code, the BDT controller's RegisterBDT method is called. For each BDT, the name of the BDT and a reference to an instance of the BDT class that implements the BDT conversion routine is passed. The BDT controller stores the BDT name and the reference in its internal table, in a similar way as it does with the Windows Registry.

Because the BDT controller is a global multi-use object, it can be invoked with its properties and methods (such as RegisterBDT) as if they were simply global functions.

Explicitly registering BDT classes in your application has the following advantages:

- You can create private BDTs that are available only inside your application. They can be developed, tested, and enhanced with your application.
- Extra keys do not need to be added to the Windows Registry to tell the BDT controller about the BDTs.

Conversion Routine

In the Construct Spectrum web framework, BDT conversion routines reside inside BDT classes.

The following table describes all of the properties and methods of the BDT Conversion object. In the examples in this table, assume the following call has been made:

```
strHours = ConvertToDisplay(dblHours, _  
    "Numeric,ZERO=OFF,ROUND=2,STRICT=ON", _  
    "N3.2")
```

Property or Method Description	
--------------------------------	--

Conversion	Tells the conversion routine what type of conversion to perform. Can be one of the following constants: bdtConvertToDisplay bdtConvertFromDisplay bdtCreateSampleString
FormattedData and RawData	When Conversion = bdtConvertFromDisplay, the conversion routine should read the value in FormattedData, convert it into a Visual Basic data type, and assign the new value to RawData.

Property or Method Description (continued)

When Conversion = bdtConvertToDisplay, the conversion routine should read the value in RawData, format it for display, and assign the formatted string value to FormattedData.

When Conversion = bdtCreateSampleString, the conversion routine should assign a sample string to FormattedData.
For example:

```
With BDT
  Select Case .Conversion
    Case bdtConvertFromDisplay
      .RawData = cvtToRaw(.FormattedData)
    Case bdtConvertToDisplay
      .FormattedData = cvtToDisp(.RawData)
    Case bdtCreateSampleString
      .FormattedData = createSample()
  End Select
End With
```

Modifier.Count Returns the number of modifiers specified by the caller.
With the example above, Modifier.Count would return 3.

Modifier Returns the value of a specific modifier or can be used to enumerate the modifiers used. Using the example above:

```
With BDT
  Print .Modifier("ZERO")      ' Prints "OFF"
  Print .Modifier(1)           ' Prints "ZERO"
  Print .Modifier(2)           ' Prints "ROUND"
  Print .Modifier(.Modifier(1)) ' Prints "OFF"
End With
```

Note: Modifier names, such as ZERO or ROUND, are passed to the BDT conversion routine in uppercase. You do not have to use case-sensitive string comparisons when checking which modifiers have been used.

Property or Method Description (continued)

FormatLength	Returns the Natural format string used in the call. Using the example above, FormatLength would return “N3.2”.
Format, Length, and Decimals	Returns the format letter, length, and decimal portions, respectively, of the Natural format string used in the call. Using the example above, Format contains “N”, Length contains 3, and Decimals contains 2.
BDTName	Returns the name of the BDT from the call. Using the example above, BDTName contains “Numeric”.
<i>Note: BDT names are passed to the BDT conversion routine with the capitalization used when the BDT was registered. For example, if RegisterBDT was called to register the BDT mIxEdCase, and then ConvertToDisplay was called for the BDT MixedCase, BDTC.BDTName would contain mIxEdCase.</i>	
SetBDTString	This method can be used to change the BDT name and the modifiers in the BDTConversion object.
ErrorCode, ErrorMessage, ErrorPos, and ErrorLen	The conversion routine should assign values to these properties if a conversion error occurred.

Creating the BDT Class

A different class module is usually used for each BDT. You can also group a set of related BDTs in a class module to share conversion routines or code. For example, a BDT called PartNumber might be implemented by a class called BDTPartNumber. You have the option to name your class whatever you wish.

The basic structure of a BDT class when implemented in Visual Basic is as follows:

```

Option Explicit

Implements IBDT

Private Sub IBDT_Convert(BDTC As BDTConversion)

    Select Case BDTC.Conversion
    Case bdtConvertToDisplay
        BDTC.FormattedData = ...
    Case bdtConvertFromDisplay
        BDTC.RawData = ...
    Case bdtCreateSampleString
        BDTC.FormattedData = ...
    Case bdtCreateSortableString
        BDTC.FormattedData = ...
    End Select

End Sub

Private Sub IBDT_SelfRegister(BDT As BDTController)
    BDT.RegisterBDT "<BDT name>", Me
End Sub

```

The BDT conversion routine is implemented in the `IBDT_Convert` procedure. It uses the properties of the `BDTConversion` object to determine what type of conversion to perform (convert to display, convert from display, create sample string, or create sortable string), to get information about the modifiers used, the Natural format specified, and to return the converted value.

You can examine the BDT classes in the `StandardBDTs` sample project to see how these properties and methods are used in BDTs.

Other BDT Controller Methods

To deregister one or more BDTs, call the `DeregisterBDT` method as follows:

```

DeregisterBDT                ' Deregisters all BDTs.
DeregisterBDT Me              ' Deregisters only the BDTs in the
                              ' specified object.
DeregisterBDT Me, "Numeric"   ' Deregisters only the Numeric BDT in
                              ' the specified object.

```

Deregistering BDTs is useful if you need to release all references to an object so that the object can be destroyed. The object can then be recreated and re-registered with all the BDTs it implements.

If the conversion routine for a given BDT needs to be located, use the `GetBDTRoutine` to return the object reference of the BDT class that implements the conversion routine. Its syntax is:

```
Sub GetBDTRoutine(ByVal BDTName As String, ByRef Handler As IBDT)
```

If the BDT name has not been registered, `Handler` will contain `Nothing` on return.

Creating a Natural-to-BDT Mapper

A Natural to BDT mapper function is called by the BDT controller when the application uses a conversion function and, instead of providing the name of a BDT, provides the Natural format. The mapper provides the most appropriate BDT to use for each Natural format.

A mapper function must be registered with the BDT controller just as BDTs are registered. Using the Windows Registry technique, the following Registry key is used:

```
HKEY_LOCAL_MACHINE
    Software
        Software AG
            Business Data Types
                NaturalBDTMapper
                    ProgID=StandardBDTs6.NaturalBDTMapper
```

You can also explicitly register the mapper function using the BDT controller's `RegisterNaturalBDTMapper` function:

```
Public Sub Main

    ...

    RegisterNaturalBDTMapper New NaturalBDTMapper

    ...

    ' Show the application's main form.
    frmMain.Show

End Sub
```

This class must implement the `IBDTMapper` interface. The following example shows how the mapper function is implemented in the `StandardBDTs` sample project:

```

Option Explicit

Implements IBDTMapper

Private Function IBDTMapper_NaturalBDTMapper(Format As String, _
                                             Length As Long, _
                                             Decimals As Integer) _
                                             As String

    Select Case Format
    Case "A"
        IBDTMapper_NaturalBDTMapper = "Alpha,TRIM=T"
    Case "B", "C"
        IBDTMapper_NaturalBDTMapper = "HexBytes"
    Case "D"
        IBDTMapper_NaturalBDTMapper = "Date"
    Case "F", "I", "N", "P"
        IBDTMapper_NaturalBDTMapper = "Numeric"
    Case "L"
        IBDTMapper_NaturalBDTMapper = "Boolean"
    Case "T"
        IBDTMapper_NaturalBDTMapper = "DateTime"
    End Select

End Function

```

The NaturalBDTMapper method must return the most appropriate BDT to use for the given Natural format/length specified.

Once a mapper function has been registered, the GetBDT method of the BDT controller returns the name of the BDT used for the given Natural format. Using the mapper listed above:

```

Print GetBDT("D")      ' Prints "Date".
Print GetBDT("L")      ' Prints "Boolean".
Print GetBDT("N6")     ' Prints "Numeric".

```

Other Consideration

The following is a consideration when creating BDTs.

One BDT Class — Multiple BDTs

You can have one BDT class implement BDT conversion routines for multiple BDTs, as the following example shows:

```
Private Sub IBDT_SelfRegister(BDT As BDTController)
    RegisterBDT "AccountNumber", Me
    RegisterBDT "DeptNumber", Me
    RegisterBDT "GroupNumber", Me
    RegisterBDT "FileNumber", Me
End Sub
```

When the application uses the BDT, the conversion routine should check the BDT name to determine what conversion to perform:

```
Public Sub IBDT_Convert(BDTC As BDTConversion)

    Select Case BDTC.BDTName
    Case "AccountNumber"
        Select Case BDTC.Conversion
            ...

    Case "DeptNumber", "GroupNumber"
        Select Case BDTC.Conversion
            ...

    Case "FileNumber"
        Select Case BDTC.Conversion
            ...

    End Select
End Sub
```

DEBUGGING YOUR CLIENT/SERVER APPLICATION

This chapter describes the procedure you can use to debug client/server applications created with Construct Spectrum.

The following topics are covered:

- **Overview**, page 202
- **Types of Errors**, page 206
- **Generating Debug Data**, page 208
- **Running Spectrum Dispatch Services Online**, page 217
- **Simulating Client Calls to Use Natural Debugging Tools**, page 219
- **Debugging Tools on the Client and Server**, page 223
- **Troubleshooting Quick Reference**, page 229

For related information, see:

- *Construct Spectrum Messages* for a list of each Construct Spectrum error with possible causes and solutions.
- *Natural Utilities Manual*
The **Debugging** section in Chapter 4, **Natural Test Utilities**, provides a detailed discussion of each of the functions of the Natural Debugging facility.
- *Microsoft Visual Basic Programmer's Guide*
Chapter 16, **Debugging**, discusses the debugging environment for Visual Basic applications including kinds of errors, different modes, and the debugging tools available.

Overview

Client/server applications are more complex than traditional, single-platform applications. Multiple computers are connected together, requiring a communication layer that opens the door for new types of errors. In client/server applications, errors can occur in more than one place. Server components of client/server applications must be developed as callable routines without a user-interface. Data values have different internal representations on the client and on the server. All of these distributed computing issues for client/server applications allow more room for errors.

Because it is not always apparent where the errors occur, debugging client/server applications can be more difficult than debugging single-platform applications. Errors may occur within the client software, the server software, the network layer, or a combination of these. To simplify the debugging process, the Construct Spectrum client framework provides you with tools and procedures that you can follow to make debugging your applications easier.

Communication Errors

Communication errors occur during a remote call from the client application to the subprogram. Each remote call involves many individual software components and data files. Some software components run on the client, while others run on the server. With so many components and different platforms involved in every call, the potential for error is greater than in non-client/server applications.

A high-level list of the components involved in a remote call follows:

- application service definitions
- client application
- EntireX Broker
- EntireX Broker stub
- Entire Net-Work
- library image files
- Spectrum Dispatch Client
- Spectrum dispatch service
- Spectrum security service
- subprogram proxy
- subprogram

Communication Error Handling

Because the client application initiates every remote call, it is also necessary to transfer back to the client application any error that does occur. The client application takes corrective action or it displays the error message to the user.

Error messages return to the client application in all but the most severe error situations. The Spectrum Dispatch Client makes the error details available to the client application through its error properties `ErrorMessage`, `ErrorNumber`, `ErrorSource`, and `ErrorValue`. If `DisplayErrors` is set to `True`, the Spectrum Dispatch Client will also display the error message in a message box.

Severe error situations that prevent the error message from being returned to the client application include:

- An interruption in the Entire Net-Work communication between the client and the server
- EntireX Broker ends
- EntireX Broker times out during the subprogram execution
- the subprogram or one of the Spectrum services causes the Spectrum dispatch service to end

If a message cannot be returned to the client, it is written to the communication log.

For a complete list of communication errors and how to resolve them, see *Construct Spectrum Messages*.

Traditional Debugging Tools

In traditional Natural applications, logic errors are diagnosed using one of two techniques:

- temporarily adding WRITE, DISPLAY, or INPUT statements to show the contents of variables and the execution sequence of the program logic
- using the Natural Debugging facility to step through the code and the variable contents

When a client application invokes Natural services, these traditional debugging tools are not available. Both of these traditional debugging techniques pause the execution of the program for user input. However, because dispatch services run in batch mode by default, no I/O statements are possible. Nevertheless, the Spectrum dispatch service may have reported Natural runtime errors or unexpected values back to the client. Each of these requires investigation.

Construct Spectrum Debugging Tools

The debugging tools supplied with Construct Spectrum allow you to:

- Save the data for client requests to a Natural library on the server. This data can then be used to recreate the request on the server and run it online. You can then use all of the traditional Natural debugging facilities to diagnose problems.

For more information, see **Generating Debug Data**, page 208 and **Simulating Client Calls to Use Natural Debugging Tools**, page 219.

- Use output statements, including WRITE, PRINT, and DISPLAY, in your Natural subprograms to write data to the Natural source buffer and save the source buffer to a Natural library. You can then examine this data after the call returns to the client. Use this technique if you do not need to run client requests online.

For more information, see **Generating Debug Data**, page 208 and **Simulating Client Calls to Use Natural Debugging Tools**, page 219.

- Examine the data transmitted between the client and the server.

For more information, see **RequestProperty Properties**, page 233.

- Examine the data expected by a subprogram proxy. Use this feature if you suspect the data formats used by the client and server components differ.

For more information, see **Diagnostics Program**, page 223.

Types of Errors

Errors that are returned by the Spectrum Dispatch Client fall into two categories: runtime errors and communication errors. A third category, Spectrum system messages, are not returned to the Spectrum Dispatch Client. These messages must be viewed in the Spectrum Administration subsystem.

While most errors can be fixed on the client, others must be fixed on the server. Construct Spectrum provides methods that help you track the origin and reason for errors. These methods allow you to determine what needs to be fixed and where the repair must be made. The types of errors you will encounter while designing your Construct Spectrum client/server application are:

- Visual Basic runtime errors
- communication errors
- Natural runtime errors
- Construct Spectrum-related errors
- errors that do not return an error message

This chapter describes the Construct Spectrum tools and procedures to help debug these last three types of errors: Natural runtime errors, Construct Spectrum-related errors, and errors that do not return an error message.

Visual Basic Runtime Errors

Visual Basic runtime errors can be trapped by using the Visual Basic On Error statement. These errors are the easiest to resolve because they occur in your Construct Spectrum application in Visual Basic and allow you to use the Visual Basic-provided debugging features to pinpoint the problem. Runtime errors are always caused by programming errors in your code or by some problem related to the client environment, such as a missing file.

Note: You also code business validations in your Visual Basic maintenance objects to raise runtime errors when a validation fails. The Construct Spectrum client framework traps these errors and displays them as pop-up messages attached to a GUI control.

*For more information about validating your data, see **Validating Your Data**, page 327, Developing Client/Server Applications. For a complete list of runtime errors and how to resolve them, see *Construct Spectrum Messages*.*

Communication Errors

Communication errors occur when there are problems establishing a connection to the server. These errors are returned by the Spectrum Dispatch Client's error properties. If `ErrorSource` contains "ETB", a communication error has occurred.

For more information, see *Construct Spectrum Messages*. You may also choose to consult the *EntireX Broker Error Reference Manual*.

Natural Runtime Errors

Natural runtime errors may occur in your subprograms. These errors are always returned to the client application by the Spectrum Dispatch Client. When the client application uses the `CallNat` method of the Spectrum Dispatch Client's dispatcher object to call a remote subprogram and the `CallNat` is returned, check the dispatcher object's error properties. If `ErrorSource` contains "NAT", a Natural runtime error has occurred.

Construct Spectrum-Related Errors

These errors are returned by the Spectrum Dispatch Client. If `ErrorSource` contains "SPE", a Construct Spectrum-related error has occurred.

For more information, including a complete list of Construct Spectrum errors and how to resolve them, see *Construct Spectrum Messages*.

Errors that Do Not Return an Error Message

These errors do not return an error message, but they can cause your program to behave unexpectedly.

Generating Debug Data

Generating debug data is a service provided by the Spectrum dispatch service. The Spectrum dispatch service automatically saves the source area contents to the Natural system file. The source area's contents are generated based on values found in the trace options set on the client. Values assigned in your user record determine the location and name of the stored debug data.

For more information, see **Specifying Where Debug Data Should Be Saved**, page 213.

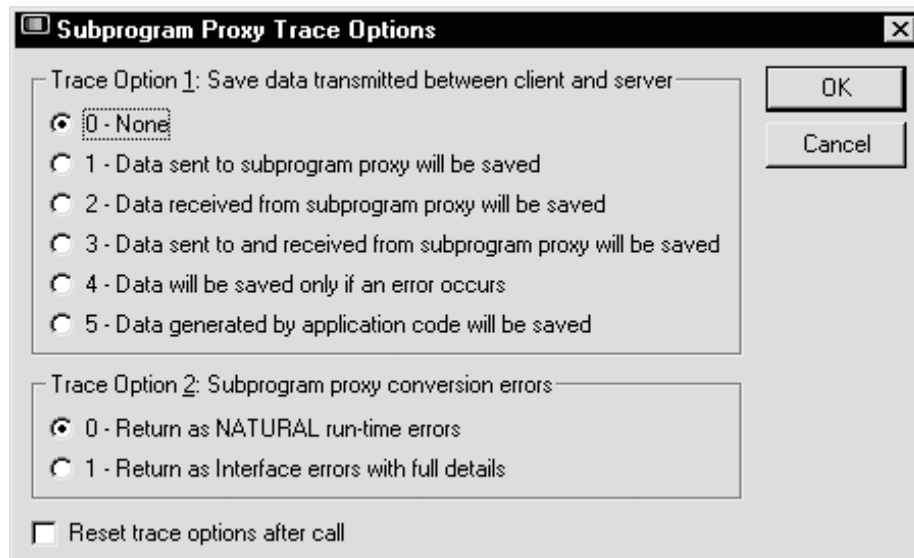
Saving Parameter and Debug Data

For each request handled by the Spectrum dispatch service, it is possible to save the parameters passed in or out of the subprogram proxy. These parameter values are saved to a text member within the Natural System file. It is also possible to save data that the application code generates into the source area.

The Spectrum Dispatch Client and Spectrum dispatch server support a trace option that determines how much debug data is saved during a remote CallNat. The trace option is set on the client before issuing the CallNat method. The Spectrum dispatch server then examines the trace option during the CallNat to determine how much data to save.

Setting Trace Options

- To set a trace option:
 - 1 Place a break point in the Visual Basic code just before the Dispatcher.CallNat method.
For a maintenance dialog, it will be in the InvokeRemoteMethod function of the Visual Basic maintenance object. For a browse dialog, it will be in the CallDBLayer function of the BrowseBase class.
 - 2 Enter "SetTraceOptions Dispatcher" in the Visual Basic Debug window where Dispatcher is the reference variable of a Dispatcher object.
Construct Spectrum displays a Trace Options dialog that allows you to set the trace options:



Remote Dispatch Server Trace Options Dialog

Trace-Option(1)

Trace-Option(1) controls how you save data to the Natural System file.

Use of Trace Option(1) causes the Spectrum dispatch service to issue an END TRANSACTION command. As a result of the END TRANSACTION, the current data is saved to the debug file.

Trace-Option(1) can be assigned one of the following values:

Value Assigned	Result
0	No tracing. Nothing will be written to the Natural system file.
1	The Spectrum dispatch service saves only data received from the client and sent to the subprogram proxy and writes it to the system file.

Value Assigned	Result (continued)
2	The Spectrum dispatch service saves only data received from the subprogram proxy and returned to the client and writes it to the system file.
3	The Spectrum dispatch service saves both the data received from and the data returned to the client.
4	The server saves data only when a Natural runtime error occurs within the server application. The data saved will be the contents of the subprogram proxy parameters at the time the error occurred. These values may differ from the values originally sent to the subprogram proxy.
5	Any data that the subprogram proxy or the subprogram writes to the Natural source area will be saved.

Note: When you are using trace option(1) = 3 and a subprogram that clears the source area is called, data received from the client is lost. Only data transmitted to the client is saved. In this case, use value 1 to save data received from the client.

If a subprogram writes data to a source area, it is then saved by the dispatcher. To write to the source area, the application's subprograms must contain a printer definition such as `DEFINE PRINTER (DEBUG=1) OUTPUT 'SOURCE'`. The subprogram can then write out debug data using Natural's `DISPLAY`, `WRITE`, and `PRINT` statements.

By default, all generated subprogram proxies contain a printer definition allowing debug data to be written to the source area. This eliminates the need for you to place this code in the generated proxy if you need to allow generation of application debug data from inside the generated proxy routine.

To write debug data to the source area, you will write code to the beginning of the module that will write the debug data (the `START-OF-PROGRAM` user exit if using Construct-generated code). To see a sample of this default and tailored code, see **Creating Debug Data**, page 211.

For more information about the subprogram proxy, see **Using the Subprogram Proxy Model**, page 129.

Creating Debug Data

The following example shows code samples of how to include debug information in your applications, and code samples of what you might see returned.

Example of code to write data to the source area

```
IF *LEVEL EQ 1 THEN
  DEFINE PRINTER(DEBUG=1) OUTPUT 'SOURCE'
END-IF
FORMAT(DEBUG) PS=0 LS=250 SG=OFF ZP=OFF AD=Z
```

To create better, more readable debug information, the DEFINE PRINTER statement should be bounded by an IF condition that does not execute in application subprograms. The DEFINE PRINTER statement is still required in each module that is expected to perform WRITE statements in the source area. However, based on the IF statement, the code is never executed; it only exists to allow for the definition of a logical printer name for the debugging target. By disallowing execution of the DEFINE PRINTER statement in application code, the print queue remains open across all subprograms using it. Each DEFINE PRINTER closes the print queue. While no information is lost, a new page header is forced each time, causing less readable debug data to be produced.

Example of debug code in a series of subprograms

```
Subprogram 1 (SUBP1)
WRITE *PROGRAM

Subprogram 2 (SUBP2)
WRITE *PROGRAM

Subprogram 3 (SUBP3)
WRITE *PROGRAM

Subprogram 4 (SUBP4)
WRITE *PROGRAM
```

Results when debugging using the IF statement

```
SUBP1
SUBP2
SUBP3
SUBP4
```

Results when debugging without using the IF statement

```
*/
Page 1

SUBP1
*/
Page 1

SUBP2
*/
Page 1

SUBP3
*/
Page 1

SUBP4
```

Use any output statement to generate information into the source area:

```
WRITE (DEBUG) NOTITLE 'Prompt 1:' #VAR1
```

or

```
PRINT (DEBUG) NOTITLE 'Prompt 2:' #VAR2
```

or

```
DISPLAY (DEBUG) NOTITLE 'Prompt 3:' #VAR3
```

*Note: The Natural subprograms called from the client execute in batch Natural processes. The output will go to the printer or terminal unless you redirect the output to the source area using the **DEFINE PRINTER** statement.*

For more information about using the debug data saved with TraceOption(1), see **Simulating Client Calls to Use Natural Debugging Tools**, page 219.

Trace-Option(2)

This option controls how the generated subprogram proxies handle runtime errors. It works in conjunction with the Generate Trace Code field of the subprogram proxy specification. It is used to help uncover the cause of data format and data length incompatibilities between the client and the server.

Assign one of the following values to Trace-Option(2):

Value Assigned	Result
0	All errors occurring within the subprogram proxy are handled as normal Natural runtime errors. As a result, control does not return to the Spectrum dispatch service and the current Error Transaction is invoked. The default error transaction returns a message to the client and restarts the Spectrum dispatch service.
1	Format conversion errors are trapped in an ON ERROR block of the generated subprogram proxy. A Natural runtime error does not occur for these errors, so the Spectrum dispatch service resumes control after the ON ERROR processing. If this option is used in conjunction with the Generate Trace Code parameter of the subprogram proxy, the field name and data values that triggered the error are returned to the Spectrum dispatch service and transferred to the client.

Tip: Using Trace-Option(2)=0 while running a Spectrum dispatch service online can be an effective way of determining runtime problems.

Specifying Where Debug Data Should Be Saved

Settings in your user record determine where debug information is stored and how file names are determined. User records are maintained in the Spectrum Administration subsystem.

For more information on the Spectrum Administration subsystem, see **Overview of the Construct Spectrum Administration Subsystem**, page 21 in the *Construct Spectrum Administrator's Guide*. For more information about user records, see **Defining Groups Using Natural Security**, page 100 and **Defining Users Using Spectrum Security**, page 104 in *Construct Spectrum Administrator's Guide*.

Accessing the Maintain User Table Panel

- To access the Maintain User Table panel:
 - 1 Enter "SA" in the Function field of the Construct Spectrum Administration Subsystem Main Menu.
The System Administration Main Menu is displayed.
 - 2 Enter "MM" in the Function field of the System Administration Main Menu.
The System Administration Maintenance Menu is displayed.

- 3 Enter "US" in the Function field of the System Administration Maintenance Menu. The Maintain User Table panel is displayed:

[illegible]

Maintain User Table Panel

The value in the Debug Library field is the name of the Natural library where the debug file will be saved. If no library is specified or if user information is provided by Natural security, the library name defaults to the current user ID value.

The Debug Filename can be set to one of the following values:

Value	Result
T	The Natural file name will be determined by concatenating “T” with the current time value.
U	The Debug file name is the same as the current user ID.

To view the generated debug members, use the Natural EDIT or LIST command.

Running Spectrum Dispatch Services Online

Instead of using trace option (1) with assigned value 5 (which writes to the source area), you can use a Natural session to initiate the service online. Initiating a dispatch service from a Natural session allows I/O to the terminal. This method is similar to the debugging method discussed in **Debugging Tools on the Client and Server**, page 223, but the Natural session running the dispatch service cannot perform any other tasks.

This type of dispatch Service stays active and locks control of your online Natural session until you send it a shutdown request or until it times out because of server non-activity.

To start a server online, invoke natural using the SYSSPEC profile. From the SYSSPEC library enter the following command (at the NEXT prompt):

```
'START servicename'
```

Note: You can also specify the Natural startup parameters in a Natural profile. For more information, see the Construct Spectrum Installation Manual.

Using the INPUT Statement as a Debugging Tool

If you decide to run the dispatch service online, use the INPUT statement for debugging. The INPUT statement allows you to interrupt and restart the execution of code. Use these interruptions to generate a printed copy of the INPUT statement or to copy the INPUT statement to the source area with a %C command.

If application tracing is set (trace option 1, with an assigned value 5), the dispatch service automatically writes the inputs copied to the source area into the designated debug source member.

Tip: When running the service online, bound your debug statements with `IF *USER = 'youruserID' THEN` and `END-IF` to guarantee that others using the same services do not generate any terminal I/O. As long as 'youruserID' is set to the *USER of the session in which the dispatch service has been initiated online, only your online dispatch service generates messages to the terminal.

Tip: When running a server online you can shut it down using the Broker console shutdown command. It is best to use a unique server class/server name/service to ensure that you don't shut down another server inadvertently.

Simulating Client Calls to Use Natural Debugging Tools

Debugging client/server applications can be difficult because of their distributed nature. To make the debugging process easier, Construct Spectrum includes an invoke subprogram proxy function that simulates client calls. Using this function lets you reproduce problems like runtime errors without the added complexity of communication between the client and server.

To help you use Natural to simulate client calls, the client component of applications created with Construct Spectrum can specify to the server application component that the data being transmitted must be saved on the server. Using this server-based data to drive the server component allows you access to Natural debugging techniques such as embedded INPUT or WRITE statements. In addition, by executing the server component locally on the server machine, you can use the Natural Debugging Facility.

For more information, see **Generating Debug Data**, page 208. For information on using the Natural Debugging Facility, see **Debugging** in Chapter 4, **Natural Test Utilities**, in the *Natural Utilities Manual*.

Invoking Subprogram Proxies Online

Once the debug data exists on the server, use the Invoke Proxy function in the Construct Spectrum administration subsystem to invoke the same subprogram proxy that the client attempted to call. The function uses the debug data to perform the same function the client originally requested. Once execution of the target proxy begins, one of two things can happen:

Result	Response
A runtime error occurs	The system traps this error and presents it as a message on the Invoke Proxy panel.
No runtime error occurs	The Invoke Proxy panel displays a message indicating that execution of the proxy completed successfully.

If you have added debug code to the target proxy and subprogram, the system is able to present the terminal output of these statements.

If the problem is not a runtime error, use the Natural Debugging facility to place break and watch points in the target code. You can monitor these points within the context of the Invoke Proxy function to examine variable contents and line-by-line execution.

Accessing the Invoke Proxy Function

The Invoke Proxy function is one of the options in the Construct Spectrum Administration subsystem. It is accessible through the Application Administration main menu.

For a description of how to access the Construct Spectrum Administration subsystem, see **Invoking Construct Spectrum Administration**, page 36 of the *Construct Spectrum Administrator's Guide*.

- To access the Application Administration Main Menu:
- 1 Enter “AA” in the Function field of the Construct Spectrum Administration Subsystem Main Menu.
The Application Administration Main Menu is displayed.
 - 2 Enter “IP” in the Function field of the System Administration Main Menu.
The Invoke Proxy panel is displayed:

BSSIDBGP May 08	Construct Spectrum Administration Subsystem Invoke Proxy	BSSIDBG1 09:03 AM
--------------------	---	----------------------

Debug Library: DEVDG____
 Member : DEVJM____
 DBID : ____
 FNR : ____

Direct Command: _____
 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---
 help retrn quit flip main

Invoke Proxy Panel

Using the Invoke Proxy function allows you to activate the Natural Debugging facility or to put trace statements (INPUT, WRITE, DISPLAY, or PRINT) in the Natural server modules to help diagnose the error.

By default, the system uses the FUSER or FNAT defined for the session when retrieving the debug data, depending on the library name. To use an alternate FUSER or FNAT, specify the values in the DBID and FNR fields.

Tip: Manually change the data in the debug member to generate a runtime error. Use this test either to ensure that runtime situations can be handled properly by the system or to force execution of code that occurs only in the case of runtime errors.

Debugging Tools on the Client and Server

Diagnostics Program

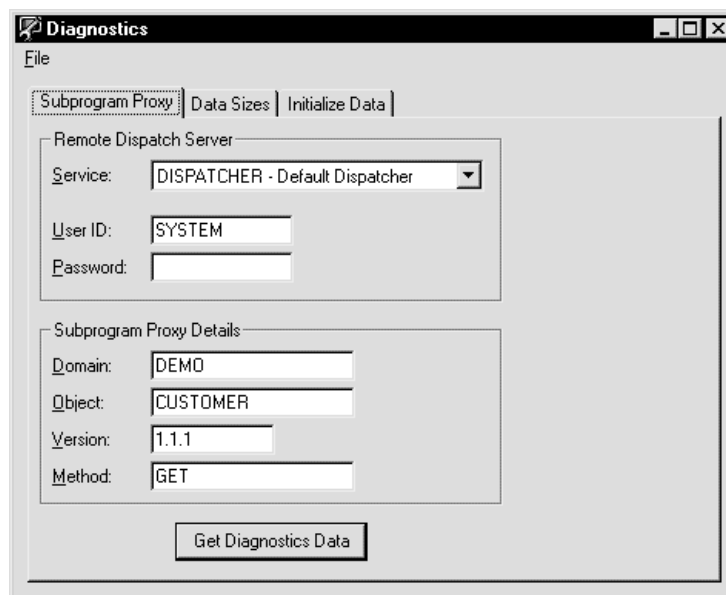
Application developers use the Diagnostics program during application development to diagnose parameter alignment problems between the client and server.

When you are invoking remote Natural subprograms from a client application, it is important that the parameters match in both size and format on both sides of the call. The Diagnostics program obtains information about the remote subprogram. By examining the Dispatcher.RequestProperty array after invoking Dispatcher.CallNat, the Spectrum Dispatch Client can give you information about the local call.

The following table summarizes the information:

Returned by the Diagnostics Program	Equivalent SDC Property
The number of level 1 blocks in the parameter data of the subprogram	Dispatcher.RequestProperty _ ("Request.DataAreas")
The name of each block; this name corresponds to the level 1 field or structure name	Dispatcher.RequestProperty _ ("Request.DataArea(2)").Name
The expanded size of the data in each block	Dispatcher.RequestProperty _ ("Request.DataArea(2)")._ PackedDataLength
The total size of the parameter data	N/A
An image of the initialized parameter data	Dispatcher.RequestProperty _ ("Request.DataArea(2)")._ PackedData

The Diagnostics program looks like this:

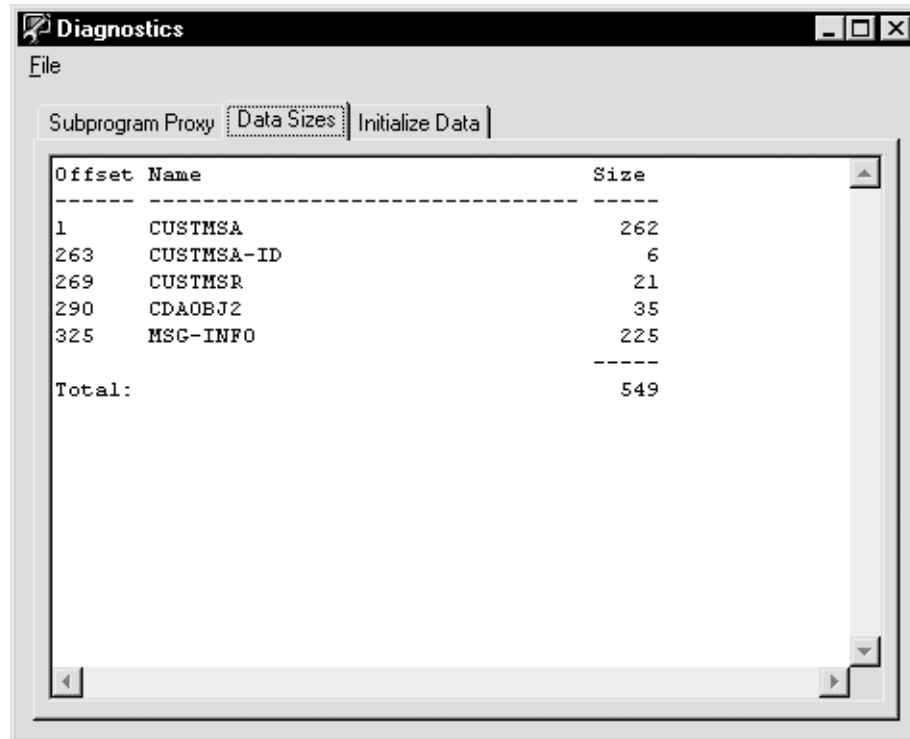


Diagnostics, Subprogram Proxy Tab

The program lets you simulate a CallNat by allowing you to provide all the Spectrum dispatch service parameters necessary to do a CallNat.

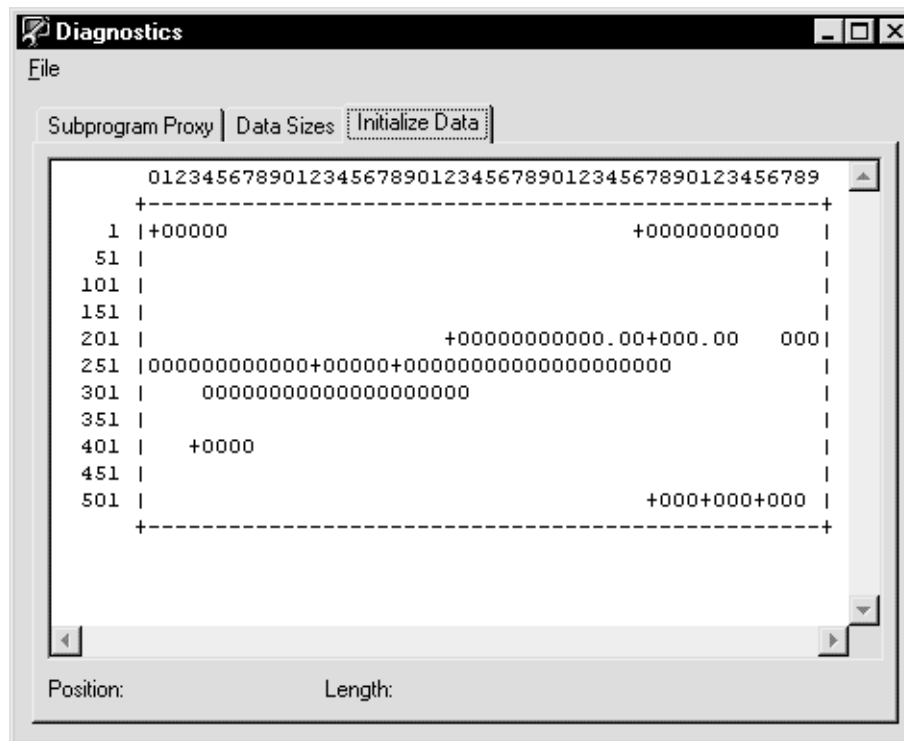
- To use the Diagnostics program:
- 1 Provide your user ID and password, which are required for all remote requests.
 - 2 Provide the domain name, object name, version number, and method name to identify the subprogram you wish to call.
You can obtain this information from the Construct Spectrum Administration subsystem or from your application's library image file (LIF).

- 3 Submit the request by clicking **Get Diagnostics Data**.
If the request is processed successfully, the **Data Sizes** tab shows information about the level 1 blocks, and the **Initialize Data** tab shows an image of the initialized parameter data. The following diagrams show these tabs for the DEMO/CUSTOMER/1.1.1/GET request shown earlier:



Offset	Name	Size
1	CUSTMSA	262
263	CUSTMSA-ID	6
269	CUSTMSR	21
290	CDAOBJ2	35
325	MSC-INFO	225
Total:		549

Diagnostics, Data Sizes Tab



Diagnostics, Initialize Data Tab

The **Initialize Data** tab shows the expanded version of the parameter data. If you highlight some text, the position and length of the highlighted section are shown at the bottom of the window. You can use this information to help determine parameter alignment problems. For example, in the diagram above, notice the first line of text on the right side of the window that reads “+0000000000”. If you know something about the format of the parameter data, you could infer that this value represents the PHONE-NUMBER field, an N10 field, in the Customer object. You can then compare the format of this data to the data sent to the server.

The Translations Program

Construct Spectrum uses its own ASCII/EBCDIC translation tables to convert data between these two character sets when the client and server use different character sets. In most cases, you do not need to know anything about these tables. However, when your subprograms send or receive non-printable characters in Alpha fields (fields with format A), you may want to know what the translation tables do with those characters.

The Translations program shows you exactly how each byte value is translated from one character set to the other. The translation tables basically group the 256 characters in each character set into three sets:

Set	Description
Printable characters	These characters exist in both character sets, and there is a well-defined mapping from one character set to the other.
Preserved characters	These characters have no corresponding character in the other character set, and their byte values are the same in both character sets. For example, character 0 in ASCII is also character 0 in EBCDIC, and 255 is 255.
Altered characters	These characters have no corresponding characters in the other character set, and their byte values are different in both character sets because the byte value is already being used by a printable character in one of the character sets.

The Translations program uses colors to identify these three sets of characters. Here is the Translation Mappings window in shades of gray:

Translation Mappings																
EBCDIC to ASCII										<input type="checkbox"/> Printable			<input checked="" type="checkbox"/> Preserved		<input type="checkbox"/> Altered	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
2	60	7F	81	82	83	84	85	86	87	88	89	91	92	93	94	95
3	96	97	98	99	A1	A2	A3	A4	A5	A6	A7	A8	A9	C0	C1	C2
4	20	[5B] 5D	C3	C4	C5	C6	C7	C8	C9	D0	. 2E	< 3C	[28	+ 2B	D1
5	& 26	D2	D3	D4	D5	D6	D7	D8	D9	E0	! 21	\$ 24	* 2A) 29	; 3B	^ 5E
6	- 2D	/ 2F	E2	E3	E4	E5	E6	E7	E8	E9	7C	, 2C	% 25	_ 5F	> 3E	? 3F
7	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	: 3A	# 23	@ 40	' 27	= 3D	" 22
8	80	a 61	b 62	c 63	d 64	e 65	f 66	g 67	h 68	i 69	8A	8B	8C	8D	8E	8F
9	90	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F	p 70	q 71	r 72	9A	9B	9C	9D	9E	9F
A	A0	~ 7E	s 73	t 74	u 75	v 76	w 77	x 78	y 79	z 7A	AA	AB	AC	AD	AE	AF
B	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C	{ 7B	A 41	B 42	C 43	D 44	E 45	F 46	G 47	H 48	I 49	CA	CB	CC	CD	CE	CF
D	} 7D	J 4A	K 4B	L 4C	M 4D	N 4E	O 4F	P 50	Q 51	R 52	DA	DB	DC	DD	DE	DF
E	\ 5C	E 1	S 53	T 54	U 55	V 56	W 57	X 58	Y 59	Z 5A	EA	EB	EC	ED	EE	EF
F	0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37	8 38	9 39	FA	FB	FC	FD	FE	FF

Translation Mappings

Troubleshooting Quick Reference

This section provides you with quick access to the most common components to check when troubleshooting.

Registry Usage

The default framework stores preferences under the following Windows registry key:

```
HKEY_CURRENT_USER
  Software
    Software AG
      CST Frameworks
```

The name of this key is set in the AppSettings.bas module. It can be changed to any other key in HKEY_CURRENT_USER.

Other framework components store application preferences in subkeys of this key.

Framework Component	Sub-Key	Specified with this Constant in CSTObjectConstants.bas
Browse	BrowseObjects	BROWSE_SUBKEY
Maintenance preferences	MaintPreferences	MAINTENANCE_SUBKEY

The following SDC preferences are stored as values under the main registry key.

Value Name	Description
DispatchService	Name of the dispatch service to use. This dispatch service name is one of the names in SDC.ini.
UserID	User ID to use in calls to the dispatch service.

SDC.ini

The SDC.ini file stores Spectrum service definitions on the client. It is located in the Windows directory. To edit the SDC.ini, use the Spectrum Service Manager.

Tip: The order of the Spectrum service definitions in this file is irrelevant.

SDCApp.ini

In the Windows directory, this file can be used to specify a dispatcher to use if there is no DispatchService entry in the registry. This functionality is not generally used because the network error dialog lets you select a dispatch definition interactively. Format:

```
[SDC]
DefaultDispatcher=<name from SDC.ini>
```

In the project directory, this file can be used to override the default LIF directory (which is the directory where the project is stored). Format:

```
[SDC]
LibraryPath=<full pathname of a LIF directory>
```

Checking for Necessary DLLs

The “ping” function of the Spectrum Service Manager is the best tool to use to check that DLLs required by Spectrum are installed and are in the path. Pinging checks for the following DLLs (in this order):

```
BROKERV.B.dll (from ETB\BIN)
CDED32.DLL (from Windows\System)
```

Construct Spectrum Add-In

The Construct Spectrum Add-In always uses the following registry key for the SDC preferences:

```
HKEY_CURRENT_USER
Software
Software AG
Construct Spectrum Add-In
```

When you do a download or upload, the Construct Spectrum Add-In uses a default library name, DBID, and FNR. It reads these from AppSettings.bas whenever you open a new project or use the Construct Spectrum Add-In for the first time in a Visual Basic session.

If you change these settings in AppSettings.bas, you will need to save the project and then restart Visual Basic to have the Construct Spectrum Add-In re-read these settings.

Visual Basic knows about the Construct Spectrum Add-In because of the following lines in Windows\VBADDIN.INI:

```
[Add-Ins32]
ConstructAddIn5.Connector=1
```

Useful SDC Properties

The SDC has many properties that you can check when you get an SDC run-time error or a communication error. Use the Visual Basic Debug Immediate to examine these properties. Some of these are displayed in the Network Error dialog box.

Application Object

Property	Description
LIFDirectory	Directory where the SDC looks for LIF files. Defaults to project directory (or when running an EXE, where the EXE is located). May be overridden with the SDCApp.ini file in the same directory.
MainLibrary	Name of the main LIF file. Set in AppSettings.bas, with the DefaultLibrary variable.
UserID	User ID to use in calls to the dispatch service.

NaturalDataArea Object

Property	Description
LibraryImageFile	Name of the LIF file that the data area definition was loaded from.
Definition	Data area definition as read from the LIF file.
PackedData	Wire-buffer representation of the field values in the data area.
PackedDataLength	Size of the wire buffer representation (in characters).
Name	Name of the data area. The name may contain other components if this data area was created by using the FieldRef method of another NaturalDataArea object, or if the data area contains 1:V fields.

Dispatcher Object

Property	Description
ErrorSource	One of ETB, NAT, or SPE
ErrorNumber	The error number, formatted according to the error source
ErrorMessage	The error message
ErrorValue(<i>i</i>)	A substitution parameter for the message

RequestProperty Properties

The SDCLib.Dispatcher object has a property called RequestProperty that returns information gathered during the last CallNat. Its syntax is:

```
RequestProperty(PropertyName As String) As Variant
```

The following property names are defined (the last column indicates whether this property is shown in the Network Error dialog box):

Value Name	Data Type	Description	X
Request.AppService	String	Name of the application service definition (e.g. CUSTOMER). This is the first parameter of the CallNat method. This application service definition is looked up in the LIF files.	X
Request.DataAreas	Integer	Number of NaturalDataArea parameters passed into the CallNat method.	X
Request.DataArea(<i>i</i>)	NaturalDataArea	1 <= <i>i</i> <= Request.DataAreas. Returns a reference to a NaturalDataArea object passed into the CallNat method.	X

Value Name (continued)	Data Type	Description	X
Request.BlocksOut	String	Block header and data blocks (in wire-buffer format) passed to the subprogram proxy.	
Request.BlocksIn	String	Blocks header and data blocks (in wire buffer format) received from the subprogram proxy.	
Request.Domain	String	Domain name read from the application service definition.	X
Request.Object	String	Object name read from the application service definition.	X
Request.Version	Long	Version number read from the application service definition.	X
Request.Method	String	Method name specified in the CallNat, or DEFAULT if not specified. The number of blocks and the blocks to send are looked up in the application service definition.	X
Request.InsideTransaction	Boolean	True if StartTransaction has been called. All requests will be sent to a dedicated dispatcher.	X
Request.DataOut	Byte Array	Full dispatcher request, starting with the request protocol bytes and the format byte.	

Value Name (continued)	Data Type	Description	X
Request.RawDataIn	Byte Array	Binary response data received from the dispatcher, after all packets have been assembled. Starts with the response protocol bytes and format byte.	
Request.DataIn	Byte Array	Dispatcher response message, after decryption, expansion, and translation to ASCII. Starts with the 4-digit dispatcher message number.	
Request.ReceivedData	String	Request.Data converted to a String with the function StrConv(Request.DataIn, vbUnicode). If the dispatcher message number is “0000”, contains the same as Request.BlocksIn.	
Packet.CountOut	Integer	Number of packets sent to the dispatcher.	
Packet.DataOut(<i>i</i>)	Byte Array	(<i>i</i>) can be from 1 to Packet.CountOut. The bytes sent for packet <i>i</i> .	
Packet.CountIn	Integer	Number of packets received.	
Packet.DataIn(<i>i</i>)	Byte Array	(<i>i</i>) can be from 1 to Packet.CountIn. The bytes received for packet <i>i</i> .	
ETB.ConversationsID	String	Broker control block values for the last Broker call.	

Value Name (continued)	Data Type	Description	X
ETB.Token	String	Generated to be unique.	
ETB.UserID	String	Always "SPECTRUM-DISPATCH-CLIENT".	

DEPLOYING YOUR CLIENT/SERVER APPLICATION

Once a Construct Spectrum project is developed and tested, the new application can be copied and installed on as many client machines as required. This chapter gives you an overview of the different considerations you need to keep in mind when deploying your client/server application.

To copy definitions from your test or development file on the mainframe to your production environment, you can use the data transfer utilities, or you can do this manually.

The following topics are covered:

- **Data Transfer**, page 238
- **Distributing Your Application**, page 239

For related information, see **Deploying the Construct Spectrum Administration Subsystem**, page 157, *Construct Spectrum Administrator's Guide*.

Data Transfer

Your test and development files may differ from the production environment. To assist you in copying definitions use the data transfer utilities. You can also manually transfer this information by using the Construct Spectrum Administration subsystem.

Data Transfer Utilities

To copy definitions quickly, use the data transfer utilities. You can copy domains and groups between one Spectrum system file and another using these utilities.

For more information, see **Data Transfer Utilities**, page 165, *Construct Spectrum Administrator's Guide*.

Manual Data Transfer

You have the option, prior to deploying your application, of manually defining and maintaining the domain, application service definitions, and steplib information in the Construct Spectrum Administration subsystem.

For information on grouping application objects and services, see **Define a Domain**, page 54.

For information on defining the domain, business object, and version of a Visual Basic business object, see **Accessing the Maintain Application Service Definitions Panel**, page 141.

For more information about identifying where your application libraries reside on the server, see **Define a Steplib Chain**, page 52.

Distributing Your Application

This section describes the procedure required to package the client application and install it on another PC. Follow these steps to distribute your application.

There are four main steps to follow when distributing your client application:

- 1 Creating the executable
- 2 Collecting files for installation
- 3 Installing the client application
- 4 Running the application

Step 1 — Creating the Executable

The first step is to create the executable. On the Visual Basic **File** menu, choose **Make EXE File**.

Step 2 — Collecting Files For Installation

Collect each of the following files for installation onto the target PC:

- the executable created in the previous step
- all runtime support files required by the executable
- the library image files, which you will install in the same directory as the executable file
- any resource files your application accesses from the client framework's Resource class and any sound files used by validation errors
- any other data files your Construct Spectrum application uses

If the target PC already has the Visual Basic runtime support files installed, you only need to copy the executable file and the library image files onto the target PC.

This procedure of readying your files for installation differs depending on whether you are creating an installation tape, installing from disk, or using a server to copy your files onto the target PC.

You may choose to use Visual Basic's Package and Deployment wizard or another setup toolkit to create a professional setup program for your application. These programs ensure that all support files required by your application are included with your setup program. You will, however, have to add the library image files to your setup program manually, just as you would for any external data file used by the application.

Warning: The Package and Deployment wizard will detect that your client application uses the Spectrum Dispatch Client and will list it as one of your application's support files. Because the Spectrum Dispatch Client is installed separately onto the target PC, you must remove the check mark from the Spectrum Dispatch Client in the list so that it is not included with your setup program.

Step 3 — Installing the Client Application

Once you have a set of distribution files, you can install the client application on the target PC.

This procedure of installing your application differs depending on whether you have created an installation tape, installation disks, or used a server to copy your files onto the target PC. It also differs depending on which setup toolkit you used to create your setup program.

There are no prerequisites for installing the client application.

Step 4 — Running the Application

Before running the client application, make sure:

- The Spectrum Dispatch Client is installed on the target PC.
- That either Entire Net-Work is installed and configured to access Entire Broker, or Entire Broker is confirmed to use TCP/IP.

You can now run your newly-installed application on the target PC. If the installation was successful, your application will behave identically to your tested application in your development environment.

If, instead, an error message is displayed, see **Debugging Your Client/Server Application**, page 201.

Note: While all error messages display on the client, some conditions can be remedied only by a system administrator.

UNDERSTANDING THE SPECTRUM DISPATCH CLIENT

This chapter describes the Spectrum Dispatch Client, a key component of Construct Spectrum. The Spectrum Dispatch Client allows you to make calls from a client to Natural subprograms running on a server.

The following topics are covered:

- **Overview**, page 244
- **Process of Creating Applications**, page 245
This section describes the four-step process required to create your applications, from creating your PDAs, assigning fields to your PDAs, using the CallNat method on the client, to checking the success of the Natural CALLNAT.
- **Spectrum Dispatch Client Components**, page 249
This section describes the Natural data area simulation, the client/server communication, the library image files, and the steplib chain.
- **Advanced Features**, page 287
This section describes how to use the advanced features of the Spectrum Dispatch Client.

For related information, see:

- **Creating Spectrum Applications Without the Client Framework**, page 297 of this guide.
This chapter describes the process of creating applications using Construct Spectrum without using the client framework.

Overview

The purpose of the Spectrum Dispatch Client (SDC) is to give application developers the ability to make calls from a client to Natural subprograms running on a server.

To create your Construct Spectrum application, follow the series of steps described in this chapter. The following small example illustrates the process you will learn:

Example of Natural Subprogram CUSTN with Defined Parameter Data

```
DEFINE DATA
  PARAMETER USING NCUSTPDA
  PARAMETER USING NCUSTPDR
  PARAMETER USING CDAOBJ
  PARAMETER USING CDPDA-M
  ...
END-DEFINE
```

Example of Natural Code that Calls Subprogram CUSTN

```
DEFINE DATA
  LOCAL USING NCUSTPDA
  LOCAL USING NCUSTPDR
  LOCAL USING CDAOBJ
  LOCAL USING CDPDA-M
END-DEFINE
*
ASSIGN NCUSTPDA.CUSTOMER-NUMBER = 10001
ASSIGN CDAOBJ.#FUNCTION = 'GET'
CALLNAT 'CUSTN' NCUSTPDA NCUSTPDR CDAOBJ CDPDA-M
...
END
```

With the Spectrum Dispatch Client, you can write similar Visual Basic code that declares these Natural data areas, assigns values to the fields in the data areas, performs a CALLNAT, and then examines the data areas to determine the results.

Note: Although you may choose to use any other OLE-compliant programming tool with Construct Spectrum, the exercises throughout this chapter use Visual Basic as a model for creating applications.

Process of Creating Applications

Calling a Natural subprogram from the client is broken down into four main steps:

Step	Description
Step 1	Create parameter data area instances
Step 2	Assign values to the fields in the parameter data areas
Step 3	Use the CallNat method on the client
Step 4	Check the success of the CALLNAT

The remainder of this section describes these steps.

Step 1 — Create Parameter Data Area Instances

➤ To create the PDA instances:

- 1 Declare the variables for the Natural data areas expected by your subprogram.

Example of declaring your variables

```
Dim ncustpda As NaturalDataArea
Dim ncustpdr As NaturalDataArea
Dim cdaobj   As NaturalDataArea
Dim cdpda_m As NaturalDataArea
```

In the previous example, the variable names are similar to the names of the external PDAs. For CDPDA-M, the dash character was changed to an underscore because the dash is not valid in a Visual Basic variable name.

- 2 Associate the name of the Natural data area with each variable by calling a routine that creates an instance of a Natural data area.

Example of creating an instance of a data area

```
Set ncustpda = SDCApp.Allocate("NCUSTPDA")
Set ncustpdr = SDCApp.Allocate("NCUSTPDR")
Set cdaobj = SDCApp.Allocate("CDAOBJ")
Set cdpda_m = SDCApp.Allocate("CDPDA-M")
```

The previous example calls the Allocate method of an object called SDCApp.

Note: Later you will learn how to declare and initialize the SDCApp object.

You have now created the data area instances.

Step 2 — Assign Values to the Fields in the Parameter Data Areas

- To assign values to the fields in the PDAs:
- Read and write the fields in the data areas.

Example of writing one field in each of the NCUSTPDA and CDAOBJ data areas

```
ncustpda.Field("CUSTOMER-NUMBER") = 10001
cdaobj.Field("#FUNCTION") = "GET"
```

In this example, the Field property of the NaturalDataArea object reads and writes fields.

Now you have set up the input parameters for the call.

Step 3 — Use the CallNat Method on the Client

Now use the CallNat method of a communications object called “Dispatcher” to call the remote subprogram.

Example of using the CallNat method

```
Dispatcher.CallNat "CUSTN", ncustpda, ncustpdr, cdaobj, cdpda_m
```

where:

"CUSTN"	Is the name of the Natural subprogram you wish to call
ncustpda	Are the names of the data areas that will be passed into the subprogram
ncustpdr	
cdaobj	
cdpda_m	

The syntax of the CallNat method on the Dispatcher resembles a CALLNAT in a Natural program.

Step 4 — Check the Success of the CALLNAT

Because this CALLNAT occurs between two machines over a network, an error could occur. To confirm the success of the CALLNAT, examine the Dispatcher's error properties. The Successful property of the Dispatcher object will be True if the CALLNAT succeeded. If the Successful property is False, check the ErrorNumber, ErrorSource, and ErrorMessage properties to find out what went wrong.

Example of Checking the Success of the CallNat

```
If Dispatcher.Successful Then
    ' The call was successful. Read the fields in the data areas.
    ...
Else
    MsgBox "An error occurred." & _
        " Number = " & Dispatcher.ErrorNumber & _
        " Source = " & Dispatcher.ErrorSource & _
        " Message = " & Dispatcher.ErrorMessage
End If
```

Summary

This simple example illustrates the process of calling Natural subprograms on the server machine from the client. There are, however, many other details that you must first specify before this example would run successfully. These include defining the Natural data areas (see Step 1), locating and invoking the Natural subprogram (see Step 3), and initializing the SDCApp and Dispatcher objects. These steps will be described later in this chapter.

Spectrum Dispatch Client Components

The Spectrum Dispatch Client provides the following key functions:

- Natural data area simulation
- Client/server communication

The Spectrum Dispatch Client has the following components to provide Natural data area simulation:

Component	Description
Data area definitions	Defines the fields in the Natural data areas your client applications will use
Data area allocator	Reads data area definitions and creates data area objects
Data area objects	Provide properties and methods to read and write Natural data areas in your client application

The Spectrum Dispatch Client has the following components to provide client/server communication:

Component	Description
Application service definitions	Defines all of the Natural subprograms your client application will call
Dispatch service definitions	Defines the parameters needed to communicate with a specific Spectrum dispatch service
Dispatcher objects	Provides properties and methods to interact with the Spectrum dispatch service

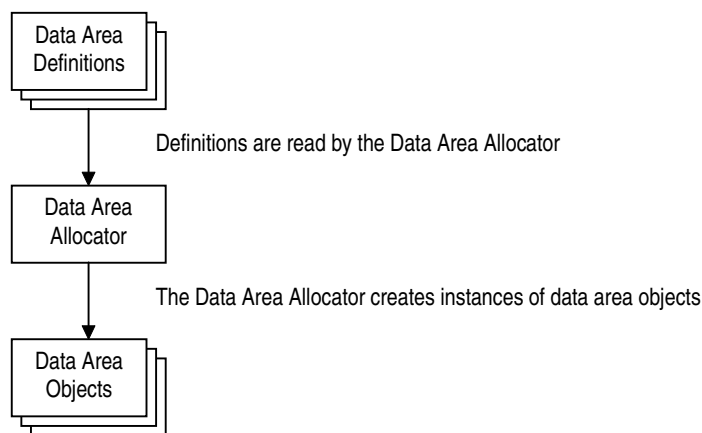
For more information, see *Construct Spectrum Messages*.

The following sections provide more information about each of these components.

Natural Data Area Simulation

When a client application calls a Natural subprogram, it passes parameters into the subprogram and receives parameters from the subprogram. Parameters are passed into and out of Natural subprograms using parameter data areas (PDAs). Construct Spectrum provides the facilities to simulate Natural data areas in Visual Basic.

The components of the Spectrum Dispatch Client that provide this capability are the data area definitions, the data area allocator, and the data area objects.



Components that Simulate Natural Data Areas

Data Area Definitions

Data area definitions use the same syntax as an inline data area in Natural code. Data area definitions are stored in library image files.

For more information, see **Library Image Files and the Steplib Chain**, page 286.

Example of the NCUSTPDA Data Area Definition

```

[DataArea NCUSTPDA]
01 CUSTOMER
  02 CUSTOMER-NUMBER (N5)
  02 BUSINESS-NAME (A30)
  02 PHONE-NUMBER (N10)
  02 MAILING-ADDRESS
    03 M-STREET (A25)
    03 M-CITY (A20)
    03 M-PROVINCE (A20)
    03 M-POSTAL-CODE (A6)
  02 SHIPPING-ADDRESS
    03 S-STREET (A25)
    03 S-CITY (A20)
    03 S-PROVINCE (A20)
    03 S-POSTAL-CODE (A6)
  02 CONTACT (A30)
  02 CREDIT-RATING (A3)
  02 CREDIT-LIMIT (P11.2)
  02 DISCOUNT-PERCENTAGE (P3.2)
  02 CUSTOMER-WAREHOUSE-ID (A3)
  02 CUSTOMER-TIMESTAMP (T)
01 CUSTOMER-ID (N5)
01 REDEFINE CUSTOMER-ID
  02 STRUCTURE
    03 CUSTOMER-NUMBER (N5)

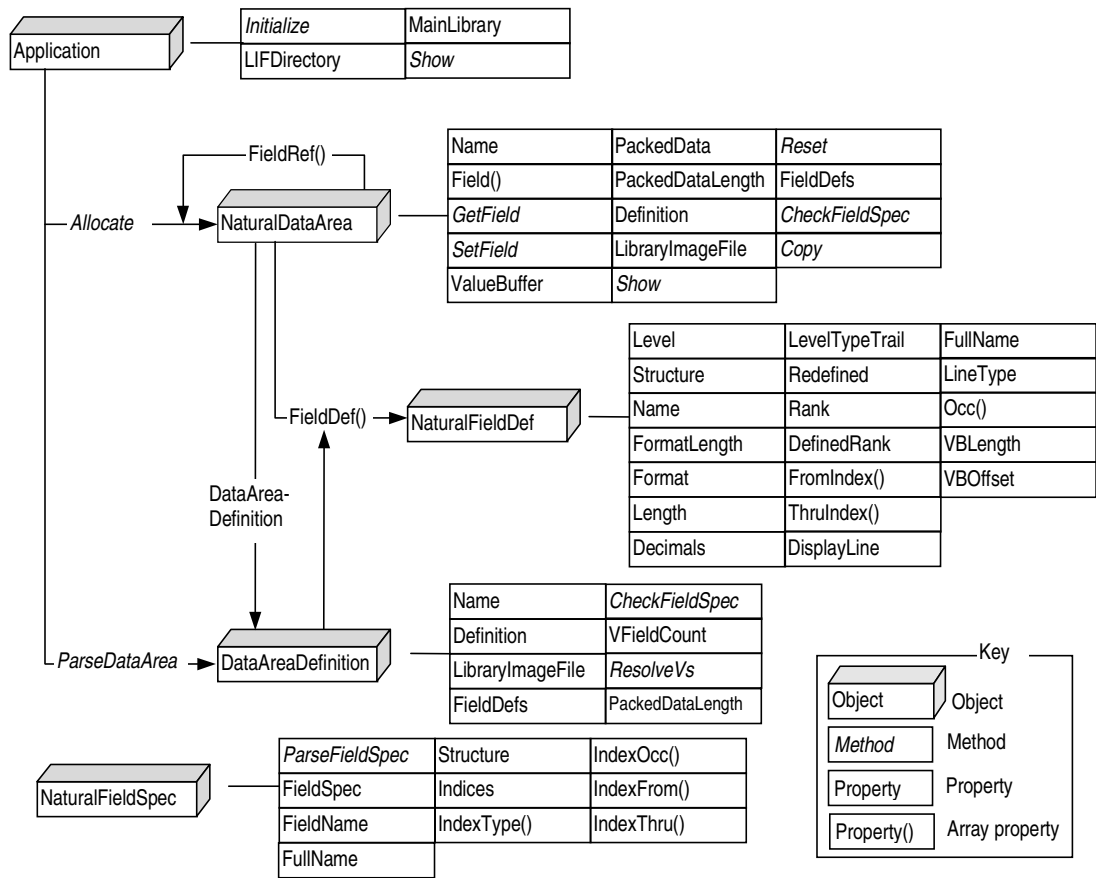
```

The Spectrum Dispatch Client supports the following features in a data area definition:

- all Natural field formats (A, B, C, D, F, I, L, N, P, and T)
- scalar fields
- one-, two-, and three-dimensional arrays
- structures
- structure arrays
- redefines, including the `FILLER` keyword
- arrays with a variable number of occurrences (1:V)

Data Area Simulation Objects

Many different Spectrum Dispatch Client objects are involved in data area simulation. These objects and their properties and methods are illustrated in the following object diagram:



Spectrum Dispatch Client Objects Involved in Data Area Simulation

The Application Object

The application object is one of the externally-creatable objects exposed by the Spectrum Dispatch Client. It has the following properties and methods that are related to data area simulation:

Property or Method Description

Initialize method	Tells the Spectrum Dispatch Client the name of the library image file directory and the name of the main library
LIFDirectory property	Returns the name of the library image file directory that was set with the Initialize method
MainLibrary property	Returns the name of the main library that was set with the Initialize method
Allocate method	Allocates a NaturalDataArea object
Show method	Displays a pop-up dialog box showing the values of all the fields in one or more data areas. The values can be edited
ParseDataArea	Similar to Allocate but simply creates a DataAreaDefinition object that can be used to discover information about a data area. The DataAreaDefinition does not store any field values.

A client application creates one global instance of the application object and uses it to create Natural data area objects.

Example of declaring and initializing the application object

```
Public SDCApp As SDCLib6.Application

Public Sub Main
    Set SDCApp = New SDCLib6.Application
    SDCApp.Initialize App.Path, "LIBRARY"
End Sub
```

This example creates a global Application object called SDCApp and then uses that object's Initialize method to set the library image file directory and the main library.

For more information on the library image file directory and the main library, see **Library Image Files and the Steplib Chain**, page 286.

Creating NaturalDataArea Objects

The Data Area Allocator reads data area definitions from library image files. It then creates NaturalDataArea objects that internally know the structure of their data area definitions. These objects allow you to read and write individual fields in their data area.

Create NaturalDataArea objects by calling the Allocate method of the application object.

The Allocate method has the following syntax:

```
Function Allocate (DataAreaName As String, _  
                  ParamArray VSubstitutions() As Variant) _  
                  As NaturalDataArea
```

where:

DataAreaName	is the name of a Natural data area.
VSubstitutions	is the parameter used when the data area has one or more 1:V arrays. For more information about data areas that contain 1:Vs, see 1:V Fields , page 293.

The NaturalDataArea Class

The data area allocator creates data area objects that are instances of the NaturalDataArea class. Each NaturalDataArea object created by the allocator internally knows the structure of its own data area definition and allows you to read and write the individual fields in the data area.

The `NaturalDataArea` class defines the properties and methods of the simulated Natural data areas. Each instance of this class stores details about its structure and maintains the field values for a single Natural data area. A client application can create as many instances of the same or different data areas as needed.

The properties and methods of the `NaturalDataArea` class are described in the following table:

Property or Method Description

CheckFieldSpec property	<p>Checks whether or not a field name is defined in the data area. Raises a runtime error if the field name is not valid. For example:</p> <pre>dataareal.CheckFieldSpec "CUSTOMER-NUMBER" dataareal.CheckFieldSpec "ROW(1) "</pre>
Copy method	<p>Creates a copy of a <code>NaturalDataArea</code> object with the same definition and the same field values. Field values changed in one do not affect the other. For example:</p> <pre>Dim data1 As NaturalDataArea Dim data2 As NaturalDataArea ' Allocate a data area. Set data1 = Nat.Allocate2 (DATAAREA_CSASTD) ' Create a copy of this data area. Set data2 = data1.Copy()</pre>
DataAreaDefinition	Provides information about the structure of a Natural data area, such as the name, format/length, and level number of each field.
Definition property	Returns a multiple-line string that contains the Entire data area definition as read from the library image file.
Field property	Used to read and write the value in a field. This property takes a field name as a parameter. If the field is part of an array, you must also specify index values as part of the field name. For example:

Property or Method Description (continued)

	<pre> With dataareal .Field("CUSTOMER-NUMBER") = 10001 .Field("PHONE-NUMBER(1)") = "4165551234" .Field("STREET(1,1)") = "134 Hill Blvd." .Field("CUSTOMER-NAME") = sname End With </pre>
FieldDef property	<p>Returns a NaturalFieldDef object that defines a field. For more information, see The NaturalFieldDef Class, page 264.</p> <p>If the field is part of an array, any provided index values are ignored. For example:</p> <pre> Dim flddef As NaturalFieldDef Set flddef = dataareal.FieldDef("M-CITY") If flddef.FormatLength = "A20" Then ... End If ' The following two lines do the same thing. Set flddef = dataareal.FieldDef("SALARY") Set flddef = dataareal.FieldDef("SALARY(1)") </pre> <p>You can also enumerate the fields in the data area by using a numeric index instead of a string field name. For example,</p> <pre> For i = 1 to dataareal.FieldDefs Print dataareal.FieldDef(i).Name Next </pre>
FieldDefs property	<p>Returns the number of field definitions in the data area definition.</p>
FieldRef property	<p>Creates a new NaturalDataArea object that contains a subset of the fields. For more information, see The FieldRef Property, page 287.</p>
GetField method	<p>Used to read the value in a field. Similar to Field, except that index values are not specified as part of the field name, but as optional parameters to this method. For example,</p>

Property or Method Description (continued)

	<pre> With dataarea1 Print .GetField("CUSTOMER-NUMBER") Print .GetField("PHONE-NUMBER", 1) Print .GetField("STREET", 1, 1) End With </pre>
LibraryImageFile property	Returns the full filename of the library image file that the data area definition was loaded from.
Name property	Returns the name of the Natural data area represented by the object. This is the name that was passed to the Allocate method.
PackedData property	<p>Returns the field values for the entire data area as an alphanumeric string. Assigning an alphanumeric string to this property replaces the field values in the data area with the values in the string. The string being assigned must be the defined length.</p> <p>For example, to copy all of the field values from one data area to another:</p> <pre>dataarea2.PackedData = dataarea1.PackedData</pre>
PackedDataLength property	<p>The length of the packed data.</p> <pre> If Len(pdata) <> dataarea1.PackedDataLength Then MsgBox "The packed data is not " & _ "the right length." Else dataarea1.PackedData = pdata End If </pre>
Reset method	<p>Resets the fields in the data area to their default values. For example,</p> <pre>dataarea1.Reset</pre> <p>You can also pass a field name into the Reset method. This will reset only that field. For example,</p>

Property or Method Description (continued)

	<pre>dataareal.Reset "CUSTOMER-NUMBER"</pre> <p>You can also reset structures and multiple occurrences of an array. For example,</p> <pre>dataareal.Reset "CUSTOMER" dataareal.Reset "STREET(*,*)" dataareal.Reset "STREET(1,*)"</pre>
SetField method	<p>Used to write the value in a field. Similar to Field, except that index values are not specified as part of the field name, but as optional parameters to this method.</p> <pre>With dataareal .SetField 10001, "CUSTOMER-NUMBER" .SetField "4165551234", "PHONE-NUMBER", 1 .SetField "134 Hill Blvd.", "STREET", 1, 1 End With</pre>
Show method	<p>Displays a pop-up dialog box showing the values of all the fields in the data area. The values can be edited. Syntax:</p> <pre>object.Show</pre>
ValueBuffer property	<p>Sets or returns a copy of the internal block of memory that stores the field values (the value buffer). This property is useful when you want to copy the field value from one data area to another. For example,</p> <pre>data1.ValueBuffer = data2.ValueBuffer</pre>

What Else Should You Know About the NaturalDataArea Object?

Field Names are Not Case-Sensitive

Field names passed into the procedures of the NaturalDataArea class are not case-sensitive. You can type the field name in uppercase, lowercase, or mixed case.

Tip: The wisest choice is to specify all field names in uppercase to be consistent with native Natural.

Alpha Fields

When reading a field with the format A, the value returned will never have trailing blanks. If a field contains only blanks, the field value will be returned as an empty string. When assigning a value to a field, the value is truncated if it is longer than the field or padded with spaces (internally) if it is shorter than the field.

Fully-Qualified Field Names

Whenever a field name is passed into the procedures of the NaturalDataArea class or DataArea, the field name can include the level 1 structure name as a qualifier. The level 1 structure name, however, is required if there is more than one field with the same name in the same data area.

Example of using the level 1 structure name as a qualifier

```
01 CDAPROXY
  02 DATA-LENGTH(I4)
  02 DOMAIN(A8)
  02 OBJECT(A32)
  02 METHOD(A32)
01 CDAOBJ
  02 OBJECT(A20)

With dataarea
  .Field("DOMAIN") = "TEST"
  .Field("CDAPROXY.OBJECT") = "EMPLOYEE"
  .Field("CDAOBJ.OBJECT") = "EMPLOYEE"
End With
```

Redefined Fields

The Spectrum Dispatch Client allows you to redefine fields, arrays, and structures just as Natural does.

Example of a Redefine

```

01 ACCOUNT (A12)
01 REDEFINE ACCOUNT
    02 COST-CENTER (A3)
    02 ACCT (A4)
    02 PROJECT (A5)

```

When the Cost-Center, Acct, or Project fields are updated, the change is also reflected in the Account field. Similarly, when the Account field is changed, the Cost-Center, Acct, and Project fields are updated.

Redefines that change the format or interpretation of data may introduce side effects that are implementation dependent.

Example of the side effects of using Redefines

```

01 OBJECT-VERSION (N6)
    02 VERSION (N2)
    02 RELEASE (N2)
    02 MAINT-LEVEL (N2)

```

```

With myver
    .Field("VERSION") = 2
    .Field("RELEASE") = 1
    .Field("MAINT-LEVEL") = 1
    Print .Field("OBJECT-VERSION")' Prints: 20101
    .Field("RELEASE") = -1
    Print .Field("OBJECT-VERSION")' Prints: <implementation defined>
End With

```

Checking Natural Fields During Compile

When you catalog a Natural module, the Natural compiler can check that fields referred to in Natural source code are actually part of the Natural data area. If the field name is not valid, if the wrong number of index values is specified for an array field, or if the data type is not compatible, the Natural compiler generates a compile error.

The Visual Basic compiler cannot check for these same errors because it does not have any knowledge of Natural. The Construct Spectrum Dispatch Client provides a *runtime* Natural simulation layer, which means that, if an invalid field name is used, the Visual Basic developer will not discover it until the statement that uses it is executed.

Reading Arrays and Structures

You must specify the necessary index values when reading or writing one-, two-, or three-dimensional arrays.

The following two examples show two different ways to read array fields:

Example of reading arrays with the GetField method

```
01 BROWSE-RECORDS(1:20)
   02 NAME(A5)
   02 OTHER-COLUMNS(A20/1:5)
01 ...

For irow = 1 To 20
    Print .GetField("NAME", irow); " ";

    For icol = 1 To 5
        Print .GetField("OTHER-COLUMNS", irow, icol); " ";
    Next
    Print
Next
```

Example of specifying a field with occurrences

```
Print .Field("OTHER-COLUMNS(" & irow & "," & icol & ")")
```

Specify the index values with a comma separating them in the Field property if the field has more than one dimension. You can also read a structure field and return it as a Byte array, as though the entire structure is defined as a B1 array. This is useful when you have a structure array and need to move occurrences of it.

The following example shows how to read and write occurrences of a structure array. This example shuffles occurrences of the Item array down to simulate deleting the occurrence number stored in the DeleteItem variable.

Example of a data area definition

```
01 ITEM(1:10)
  02 NUMBER (N5)
  02 DESCRIPTION (A30)
  02 UNIT-COST (P7.2)
  02 QUANTITY (N5)
  02 TOTAL-COST (P7.2)
```

Example of reading occurrences of the Item array

```
With dataareal
  For i = DeleteItem + 1 To 10
    .Field("ITEM(" & i - 1 & ")") = .Field("ITEM(" & i & ")")
  Next
End With
```

Runtime Errors

Many different runtime errors could result from using NaturalDataArea objects.

The DataDefinitionArea Class

The DataDefinitionArea Class provides information about the structure of a Natural data area, such as the name, format/length, and level number of each field. The NaturalDataArea and DataAreaDefinition classes have many properties in common because they both store the definition of a Natural data area. However, unlike the NaturalDataArea class, the DataAreaDefinition class does not store field values.

The Spectrum Dispatch Client provides two ways to create an instance of a DataAreaDefinition. You can use the ParseDataArea method of the Application class to parse an inline data area definition or a data area definition in an external LIF file.

Optionally, you can use the DataAreaDefinition property of a NaturalDataArea object. Internally in the Spectrum Dispatch Client, a NaturalDataArea object uses a DataAreaDefinition object to store the structure of the Natural data area. The DataAreaDefinition property returns a reference to that DataAreaDefinition object.

Property or Method Description

CheckFieldSpec property	<p>Checks whether or not a field name is defined in the data area. Raises a runtime error if the field name is not valid. For example:</p> <pre>dataareal.CheckFieldSpec "CUSTOMER-NUMBER" dataareal.CheckFieldSpec "ROW(1) "</pre>
Definition property	Returns a multiple-line string that contains the Entire data area definition as read from the library image file.
FieldDef property	Returns a NaturalFieldDef object that defines a field. For more information, see The NaturalFieldDef Class , page 264.
FieldDefs property	Returns the number of field definitions in the data area definition.
PackedDataLength property	<p>The length of the packed data.</p> <pre>If Len(pdata) <> dataareal.PackedDataLength Then MsgBox "The packed data is not " & _ "the right length." Else dataareal.PackedData = pdata End If</pre>

The NaturalFieldDef Class

NaturalFieldDef is a Spectrum Dispatch Client class that returns the definition for a single field in a data area definition. The FieldDef property of the NaturalDataArea class creates and returns an instance of the NaturalFieldDef class. All of the properties defined by this class are read-only.

The properties of the NaturalFieldDef class are defined in the following table:

Property	Description
Decimals	Returns the decimal length portion of the Natural format. If the format is not numeric (N) or packed numeric (P), this property returns 0. For example, Decimals would return 0, 0, 2, and 0 in the FormatLength property example below.
DefinedRank	Returns the number of dimensions of the field in the data area definition. This property works similar to the Rank property, except that the DefinedRank property returns the number of dimensions regardless of any structure arrays it might be part of.
Format	Returns the format character from the Natural format. For example, Format would return N, A, P, and D in the FormatLength property example below.
FormatLength	Returns the format and length of the field using the Natural syntax

Example of using the FormatLength field

```
With employee
  Print .FieldDef("PID").FormatLength
  Print .FieldDef("FIRST-NAME").FormatLength
  Print .FieldDef("SALARY").FormatLength
  Print .FieldDef("HIRE-DATE").FormatLength
End With
```

Example of one possible output of the FormatLength field (depending on the data area definition)

```
N6
A20
P8.2
D
```

Property	Description (continued)
FromIndex and ThruIndex	<p>Returns the low and high index values for each dimension of an array field.</p> <p><i>Example of data area and Visual Basic code</i></p> <pre> 01 VALUES (N10/1:10,5:7) With data.FieldDef("VALUES") For i = 1 To .Rank Print .FromIndex(i) & ":" & .ThruIndex(i) Next End With </pre> <p><i>Example of the resulting print</i></p> <pre> 1:10 5:7 </pre>
FullName	Returns the fully qualified field name (which includes the Level 1 structure name).
Length	Returns the length portion of the Natural format. If the format is D, L, or T, this property returns 0. For example, Length would return 6, 20, 8, and 0 in the FormatLength property example above.
Level property	Returns this field's level number in the data area definition
LevelTypeTrail	<p>Returns a string that can be used to determine the nesting of this field in the data area definition. This string has one character for each level. Each character can be one of the following:</p> <ul style="list-style-type: none"> F Field S Structure R Redefine X Filler

Property	Description (continued)
	<i>Example of a data area</i>
	<pre>01 ROW-COUNT (N2) 01 ROW (1:10) 02 ID (N6) 02 ACCOUNT-NO (A16) 02 REDEFINE ACCOUNT-NO 03 DIVISION (A4) 03 FILLER 1X 03 GROUP (A5) 03 FILLER 1X 03 ENTITY (A5)</pre>
	<i>Example of the results of the LevelTypeTrail property</i>
	<pre>Print .FieldDef("ROW-COUNT").LevelTypeTrail ' Prints "F" Print .FieldDef("ROW").LevelTypeTrail ' Prints "S" Print .FieldDef("ID").LevelTypeTrail ' Prints "SF" Print .FieldDef("ACCOUNT-NO").LevelTypeTrail ' Prints "SF" Print .FieldDef("DIVISION").LevelTypeTrail ' Prints "SRF" Print .FieldDef(7).LevelTypeTrail ' Prints "SRX"</pre>
Name	Returns the name of the Natural field
Occ	Returns the number of occurrences for each dimension of an array field.
Rank	Returns whether this field is a scalar field or part of an array, according to the following table: 0 Scalar

Property	Description (continued)
	1 One-dimensional array
	2 Two-dimensional array
	3 Three-dimensional array
	Rank indicates the number of index values that must be used when reading or writing the field.
Redefined	Returns True if this field is redefined further on in the data area definition.
Structure	Returns the structure name if this field is part of a level 1 structure
ThruIndex	See the FromIndex property

Client/Server Communication

The other major function of the Construct Spectrum Dispatch Client is client/server communication.

Many components work together to enable client/server communication. These include:

- Application service definitions
- Dispatcher objects
- Dispatcher service definitions

The following sections describe these components in more detail.

Before you can understand application service definitions, you must understand a feature of Construct Spectrum called level 1 block optimization.

Level 1 Block Optimization

The Spectrum Dispatch Client and the subprogram proxies implement a performance optimization feature called level 1 block optimization. This feature attempts to minimize the amount of data that is transmitted between the client and server for each remote CALLNAT.

With level 1 block optimization, each level 1 field in the parameter data of the Natural subprogram becomes a numbered block. Each block can contain one or more Natural fields, structures, or structure arrays. Instead of sending all of the parameter data between the client and server for each remote CALLNAT, the Construct Spectrum Dispatch Client and the Spectrum dispatch service are able to transmit a subset of the blocks in each direction.

To understand why this is useful, consider the following. For most Natural subprograms, each field in the parameter data can be assigned a directional attribute to indicate whether that field is used to pass data into the subprogram, out of the subprogram, or both.

Note: These directional attributes are not supported by the native Natural language. However, they may be defined in the application service definitions supported by the Spectrum Dispatch Client and coded in user exits in subprogram proxies.

The following table summarizes these directional attributes:

Directional Attribute	Description
IN	The data in this field is passed from the caller to the subprogram.
OUT	The data in this field is returned from the subprogram to the caller.
IN/OUT	The data in this field is passed from the caller to the subprogram, optionally modified by the subprogram, and then returned from the subprogram to the caller.

If the parameter data is organized such that each block (level 1 field) contains only In, Out, or In/Out parameters, then the Spectrum Dispatch Client can use level 1 block optimization to send only the In and In/Out parameters to the subprogram proxy. The subprogram proxy can send only the Out and In/Out parameters back to the client. In some cases, the size of the In or Out parameters is very small compared to the total size of the parameter data. Level 1 block optimization can make a significant difference to the size of the data being transmitted over your network.

Note: This block optimization feature does not allow directional attributes to be assigned at a level of granularity finer than level 1 fields.

Occasionally, it may not be possible to assign a static directional attribute to a parameter because it may change its direction depending on the values of other parameters. This is illustrated in the following example:

Example of parameter data for a Natural Construct object subprogram

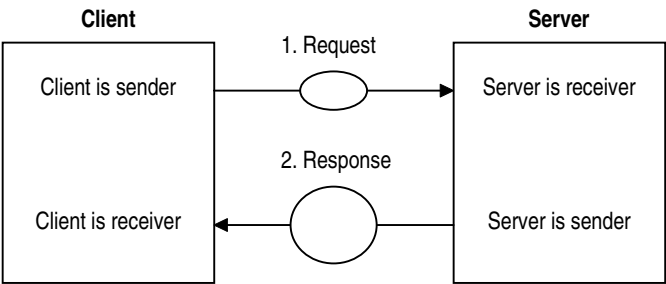
```

DEFINE DATA
    01 CUSTOMER                                /* Object PDA
        02 CUSTOMER-NUMBER (N5)
        02 BUSINESS-NAME (A30)
        02 PHONE-NUMBER (N10)
        ...
    01 NCUSTPDA-ID
        02 ...
    01 NCUSTPDR
        ...
    01 CDAOBJ
        02 #FUNCTION (A15)
        ...
    01 MSG-INFO
        02
        ...
END-DEFINE

```

The object PDA is either In, Out, or In/Out, depending on the #FUNCTION flag in CDAOBJ. When #FUNCTION contains Get, the object PDA contains data returned from the subprogram to the caller, so it is an Out parameter. When #FUNCTION contains Update, the caller is passing data in the object PDA into the subprogram, and depending on whether the subprogram performs edits on the data, the subprogram may also return updated values in the object PDA, so it is either an In or an In/Out parameter.

When using level 1 block optimization, the sender always decides which blocks will be sent to the receiver. Sender and receiver differ from client and server because the client and server are both senders and receivers.



Client and Server are Both Sender and Receiver

When the request is sent to the server, the client is the sender and the server is the receiver. When the response is sent to the client, the server is the sender and the client is the receiver.

In the previous illustration, different-sized ellipses show how the size of the request data might be different than the size of the response data because the set of blocks might be different.

Application Service Definitions

Application service definitions are defined on the server, in the Spectrum Administration subsystem, and also on the client, in a library image file. The following table compares the information stored in application service definitions on the server and on the client:

Information in an application service definition	Stored on the server	Stored on the client
Domain name	X	X

Information in an application service definition (continued)	Stored on the server	Stored on the client
Object name	X	X
Object version number	X	X
Method names	X	X
Which subprogram proxy to call for each method	X	
The steplib chain to use when calling a subprogram	X	
For each method, the number of level 1 fields in the parameter data of that method's subprogram		X
Which level 1 fields are sent to the server for each method		X

The following example shows what an application service definition looks like in a library image file:

```
[AppService CUSTOMER]
Domain=DEMO
Object=CUSTOMER
Version=1.1.1
Method=BROWSE,,4,1+3+4
Method=DEFAULT,,5,1+2+3+4+5
Method=DELETE,,5,2+3+4
Method=EXISTS,,5,2+4
Method=GET,,5,2+4
Method=INITIALIZE,,5,4
Method=NEXT,,5,2+4
Method=STORE,,5,1+4
Method=UPDATE,,5,1+3+4
```

where:

[AppService CUSTOMER]	Introduces the application service definition and identifies the name of the application service definition
Domain, Object, and Version	Identifies the corresponding application service definition in the Spectrum Administration subsystem
Method	Defines a method within the application service

Each method line contains four values separated by commas:

- A logical method name used in your Visual Basic code.
- A physical method name that corresponds to a method name in the application service definition on the server. If this name is the same as the logical method name, it can be omitted, as in the example above.
- The number of level 1 fields in the parameter data of the subprogram associated with the method. In the example above, the subprogram for the BROWSE method has four level 1 fields in its parameter data, and the subprograms for all other methods have five level 1 fields in their parameter data areas.
- Which of the level 1 fields are to be sent to the server for the method. In the example above, when calling the BROWSE method, only the first, third, and fourth (1+3+4) level 1 fields are sent to the server.

When you use the `CallNat` method on the client, you do not specify a Natural subprogram to call. Instead, specify the name of an application service definition. The SDC uses the application service definition name to look up the domain name, object name, and version number, and passes these values to the Spectrum dispatch service running on the server, which in turn uses these values to look up the subprogram proxy to call.

Thus, the application service definitions on the client and on the server work together to allow a client application to identify a subprogram on the server to call.

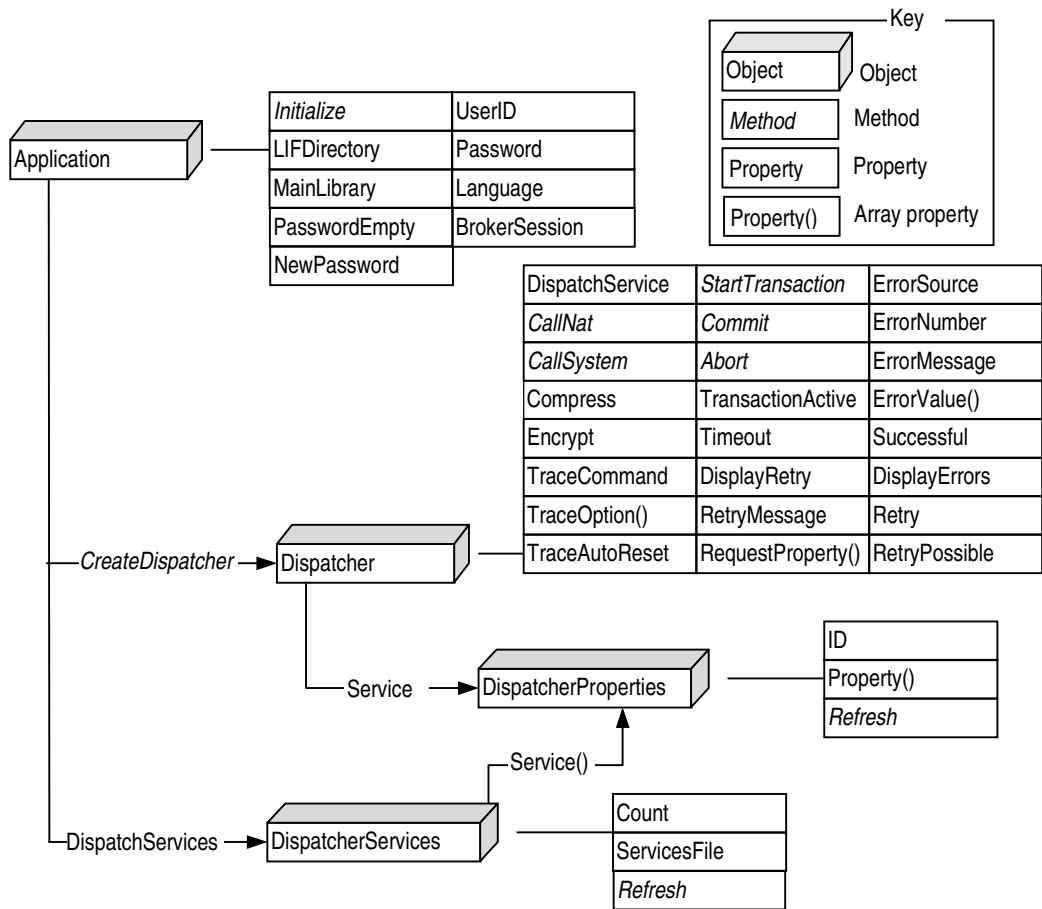
The following example shows how you might code a `CallNat` method on the client using the application service definition used above:

```
Dispatcher.CallNat "CUSTOMER.GET", ncustpda, ncustpda_id, ncustpdr, _  
                                cdaobj, cdpda_m
```

Notice how the method name “GET” is appended to the application service name “CUSTOMER”. If you do not specify a method name in the `CallNat`, the SDC uses the method name “DEFAULT”, and this method must then exist in the application service definition.

Dispatcher Objects and Dispatch Service Definitions

`Dispatcher` is a Spectrum Dispatch Client class that handles the communication between the client and the server. It can be thought of as the networking components of the Spectrum Dispatch Client. The `Dispatcher` class has many properties and methods, which are shown in the following diagram along with related objects:



Dispatcher Objects

Dispatcher objects are created with the *CreateDispatcher* method of the Application object.

Example of creating dispatcher objects

```
Dim Dispatcher As Dispatcher  
Set Dispatcher = SDCApp.CreateDispatcher()
```

The properties and methods of the dispatcher object are separated into the following functional groups.

- Service selection
- Remote subprogram invocation
- Timeout, retry, and resume handling
- Compression and encryption
- Tracing
- Database transaction control
- Error reporting

The following sections describe each of these functional groups in more detail.

Service Selection

You may have multiple Spectrum dispatch services running on one or more server platforms simultaneously. There could even be different types of Spectrum dispatch services, each with its own defaults, security settings, FUSER, and so on running at the same time. Before sending any request, the client must first identify which Spectrum dispatch service to connect to. You do this by setting the Dispatch-Service property to the ID of a valid Spectrum dispatch service.

The available dispatch services are defined in the Construct Spectrum Administration subsystem on the server platform. On the client, these dispatch services are defined with the Spectrum Service Manager.

Each dispatch service definition specifies the following values:

- EntireX Broker ID
- Server class
- Server name
- Service

If you are familiar with EntireX Broker, you will recognize that this combination of values uniquely identifies an EntireX Broker service. Each Spectrum dispatch service is actually an EntireX Broker service.

Remote Subprogram Invocation

To send a request to the Spectrum dispatch service, use the `CallNat` and `CallSystem` methods. These methods return `True` if the call was successful and `False` if the call was unsuccessful.

The `CallNat` method is used to invoke a Natural subprogram on the server.

Syntax of the CallNat Method

```
Function CallNat (ByVal AppServiceName As String, _  
                 ParamArray DataAreas() As Variant) As Boolean
```

The name of the application service is always required. Following this name, you can specify zero or more instances of the `NaturalDataArea` class that will be passed as parameters to the target subprogram. The parameters are passed by reference. Therefore when the subprogram returns, any changes that the subprogram made to the fields in the data areas will also be available in the `NaturalDataArea` objects.

To take advantage of level 1 block optimization, you can include a method name in the first parameter.

Example of implementing level 1 block optimization

```
Dispatcher.CallNat "CUSTN.GET", custpda, custpda_id, custpdr, _  
                  cdaobj, cdpdam
```

where:

<code>.GET</code>	is the method name appended to the subprogram name. Use a period character to separate the two. Only the blocks specified in the method definition will be sent to the server in the request data
-------------------	---

You will use the `CallSystem` method to send system commands to the Spectrum dispatch service or to invoke an arbitrary proxy.

Syntax of the CallSystem Method

```
Function CallSystem (ByVal DomainName As String, _
                    ByVal ObjectName As String, _
                    ByVal Version As Long, _
                    ByVal MethodName As String, _
                    ByVal SendData As String, _
                    ByRef ReceiveData As String) As Boolean
```

where:

<code>CallSystem</code>	is the method that allows you to send system commands directly to the Spectrum dispatch service or invoke an arbitrary subprogram proxy by specifying its domain, object, version, and method
-------------------------	---

Timeout, Retry, and Resume Handling

The `CallNat` and `CallSystem` methods do not return until the server has sent back a response. In effect, your calling application is locked up while the server is processing the request.

If the server does not respond, your application may not regain control and the user will have to terminate the application. For this reason, the `Dispatcher` object has a request timeout. The timeout indicates the maximum number of seconds you are willing to wait for the server to respond. When that number of seconds elapse since the request was sent to the server, the dispatcher will do one of two things:

- return control to your calling application
- or
- ask the user whether or not to continue waiting

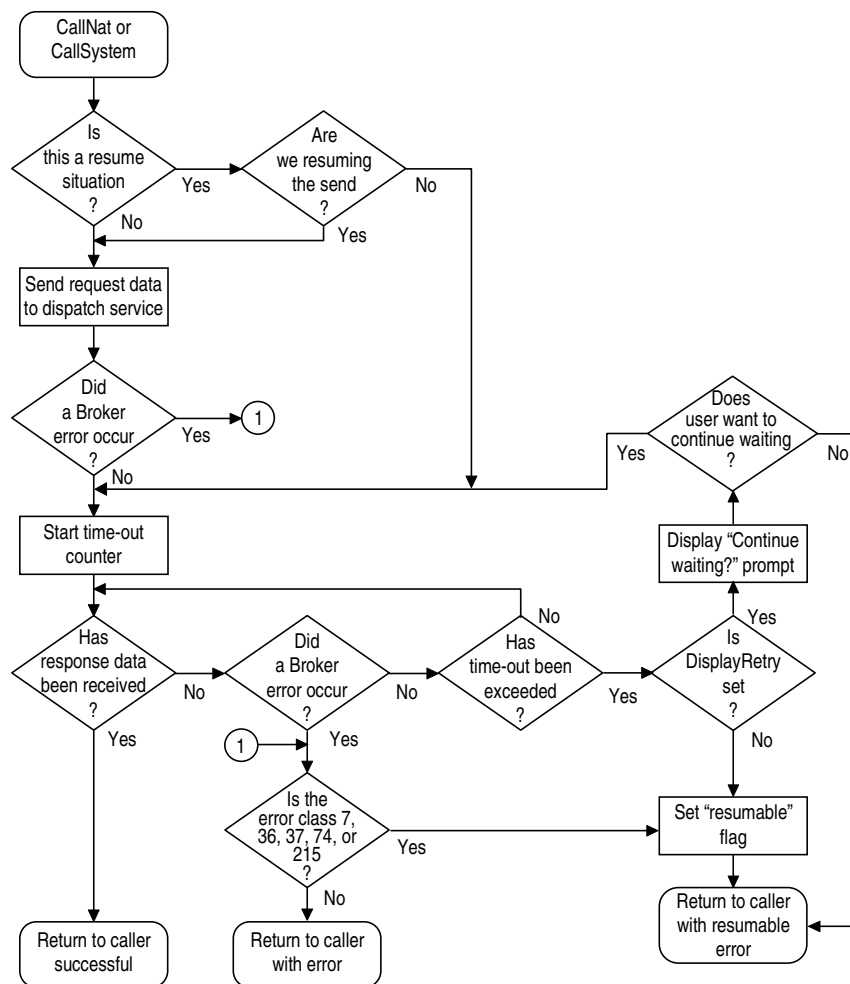
Use the `DisplayRetry` property to tell the `Dispatcher` object which of these to do. Set `DisplayRetry` to `False` to return control to your calling application. Set `DisplayRetry` to `True` to ask the user whether or not to continue waiting, and

optionally set the `RetryMessage` property to a string that will be displayed to a user. The default message is: “The server is not responding. Would you like to continue waiting?”

The `Timeout` property determines the timeout duration in seconds and can be set to any value from `-1` to `32767`. Zero (`0`) returns control to the client application immediately. Negative one (`-1`) is the default and uses the timeout value specified in the dispatch service definition.

Tip: Users can change this timeout value using the Spectrum Service Manager program. For more information about changing this timeout value, see **Using Construct Spectrum Tools**, page 139 of the *Construct Spectrum Administrator's Guide*.

The following flowchart illustrates the life-cycle of a full request and response combination:



Life Cycle of a Full Request/Response Combination Showing the Timeout Functionality

This flowchart illustrates an important feature of the Spectrum Dispatch Client:

- the ability to resume the processing of a request because of a timeout or a recoverable EntireX Broker error

Some EntireX Broker errors, such as resource shortages or a temporary interruption in Entire Network, are recoverable. If such an error occurs in the middle of processing a request, either when sending the request data to the server or receiving the response data from the server, the Spectrum Dispatch Client can return the error to the calling application, which can then decide whether to resume the request or not.

To determine if the request is resumable, check the `RetryPossible` property after returning from the call. If this property returns `True`, you may set the `Retry` property to `True` and then reissue the exact same call.

Example of resuming a call

```
Do
    If .CallNat("CUSTN.GET", custpda, custpda_id, custpdr, _
               cdaobj, cdpdam) Then
        ' Request was successful.
        Exit Do
    Else
        msg = "The following error occurred: " & _
              .ErrorSource & ":" & .ErrorNumber & " - " & _
              .ErrorMessage & vbCrLf & vbCrLf & _
              "Click OK to try again or Cancel to quit."
        If MsgBox(msg, vbOkCancel) = vbCancel Then
            Exit Do
        End If
        If .RetryPossible Then .Retry = True
    End If
Loop
```

In the above example, if an error occurs, the error message will be displayed to the user, along with a prompt asking the user whether he or she would like to try again. If the user chooses to try again, the same call is performed.

What happens during this second call depends on the setting of the `Retry` property. If the error was resumable (`RetryPossible` is `True`), set `Retry` to `True` and the previous request will be resumed. If the error was not resumable (`RetryPossible` is `False`), the second call will initiate an entirely new request.

Compression and Encryption

The Spectrum Dispatch Client can compress or encrypt the request data it sends to the Spectrum dispatch service.

Compression can significantly reduce the size of the data, which can in turn reduce the transmission time, especially over slow network connections such as dialup connections. The compression algorithm reduces sequences of repeating characters, which are quite common when the request and response data contain partially-filled Natural data areas.

To enable compression, set the Compress property to True. To enable encryption, set the Encrypt property to True. These properties remain set until you change them.

Note: These properties only compress and encrypt the request data, sent from the client to the Spectrum dispatch service. The decision to compress or encrypt the response data is made in the subprogram proxy on the server platform.

Tracing

Tracing options provide you with the opportunity to track the data transmission to and from the server. You will set these tracing options, depending on the type of data you would like to trace, by setting the properties of the Dispatcher object.

The following Dispatcher object properties are available to set trace options:

- TraceOption, an array property with indices 1 to 15
- TraceCommand, a string property
- TraceAutoReset, a boolean property which automatically resets the trace options after the call to the Spectrum dispatch service.

For more information about setting tracing options and understanding the result, see **Debugging Your Client/Server Application**, page 201.

Database Transaction Control

Each request you send to the Spectrum dispatch service could be handled by a different copy of the Spectrum dispatch service. While you are processing a request, you have exclusive access to the server. However, once the server sends the response data back to the client PC, the server is available for your next request or for a request sent by someone else.

The Construct Spectrum Dispatch Client also gives you exclusive access to a specific server across more than one request. To have this exclusive access, you must specify when you want to start having exclusive access to a server and when you are finished with it. While you have exclusive access, the server is dedicated to your client application and will only accept requests from you. No other client application can send requests to that server (unless you pass a reference to the Dispatcher object to another client application). Therefore, you should try to release the server as soon as possible because you are preventing anyone else from using the server, and there may only be a limited number of servers running.

When you have exclusive access to a server, you can also issue `END TRANSACTION` or `BACKOUT TRANSACTION` statements from the client application and be assured that only your requests will be affected.

The Dispatcher class has three methods and one property to support exclusive use of a server.

Method or Property Description

StartTransaction method	Tells the Dispatcher object that you want to have exclusive access to a server
Commit method	Sends a special request to the server to issue an <code>END TRANSACTION</code> statement and releases the server
Abort method	Sends a special request to the server to issue a <code>BACKOUT TRANSACTION</code> statement and releases the server
TransactionActive property	Returns True if you have exclusive access to a server

Each Spectrum dispatch service has a transaction timeout value that ensures a client application does not have exclusive access to the server for too long. The transaction timeout period begins as soon as the server has sent the response data back to the client application. If the client application does not send any more requests to the server within the timeout period, the server will automatically issue a BACKOUT TRANSACTION statement and return to the server pool. If this happens, the client application is not notified until it tries to send the next request. It will then fail with a `sdcerrTransactionTerminated` error.

Note: The transaction timeout period is set in the Construct Spectrum Administration subsystem in the Maintain Services panels.

To prevent transaction timeout, you should try to send all requests in succession and then release the server. If your application interacts with the user between requests (or if an error occurs and you display it to the user), there will be a greater possibility of transaction timeout occurring because the user may not respond immediately.

The server is also automatically released when the Dispatcher object is destroyed (after all object references to it have been released).

Error Reporting

The errors that can occur in the Dispatcher object fall into two categories: runtime errors and communication errors.

Error Types	Description
Runtime errors	Raised using the standard OLE automation error handling mechanism
	For more information about runtime errors, see Deploying Your Client/Server Application , page 237 or <i>Construct Spectrum Messages</i> .

Error Types	Description
Communication errors	occur during a remote call from the client application to the subprogram and the error details are returned in the error properties of the Dispatcher object, namely ErrorSource, ErrorNumber, ErrorMessage, ErrorValues, and Successful. For more information about communication errors, see Deploying Your Client/Server Application , page 237 or <i>Construct Spectrum Messages</i> .

User Identification and Authentication

The Application object has properties for UserID, Password, and Language. These properties must be set before the first request is sent to the server, but they may be changed at any time after that.

Application Properties	Description
UserID	Identifies who you are to the server
Password	Ensures you are who you say you are
PasswordEmpty	Returns whether or not the Password property has been set
Language	Used by servers that have been internationalized

If the server uses security, it can authenticate your userID and password and then check whether or not you have the necessary permissions to execute the request. The dispatcher will check your permissions for every request. If the server does not use security, any userID and password you assign to these properties will be ignored.

Use the Language property to indicate your spoken language by assigning one of the *LANGUAGE codes defined by Natural to this property. This code is sent to the server with each request. Whenever the server needs to return a message string to you, the server will look up the correct translation based on the language code.

Library Image Files and the Steplib Chain

Library image files are special text files that contain Spectrum Dispatch Client definitions. Each library image file contains up to three different types of definitions:

- Data area definitions
For more information, see **Data Area Definitions**, page 250.
- Application service definitions
For more information, see **Application Service Definitions**, page 271.
- Steplib definitions

Syntax of the steplib definition

```
[StepLibs]  
CST411S  
SYSTEM
```

A steplib definition allows multiple applications to share a set of library image files (LIFs). Each application may have its own main library, which contains just the definitions specific to that application. Shared definitions can be placed in other library image files, which can be included in each application's steplib chain.

When searching for data area and Application Service Definitions, the Construct Spectrum Dispatch Client first examines the main library's library image file. If it does not find the definition there, it looks for a steplib definition in the file. If it finds the steplib definition, it will examine the library image files for the libraries in the steplib definition, beginning with the first LIF in the list.

Advanced Features

This sections introduces two advanced features you will turn to as you are developing your applications. It includes:

Feature	Description
FieldRef Property	Allows you to define objects as parameters without duplicating data areas to pass objects through to a Natural CALLNAT
1:V	Allows you to define arrays with variable numbers of occurrences

The FieldRef Property

The CallNat method of the Dispatcher class only accepts objects of type NaturalDataArea as parameters to pass to subprograms.

You can, however, pass individual fields into a subprogram in real Natural code.

Example of passing individual fields into a subprogram

```
ASSIGN CBROWSEA.COUNT = 10
CALLNAT 'CUSTB' CBROWSEA.COUNT
          CBROWSEA.ROWS (*)
```

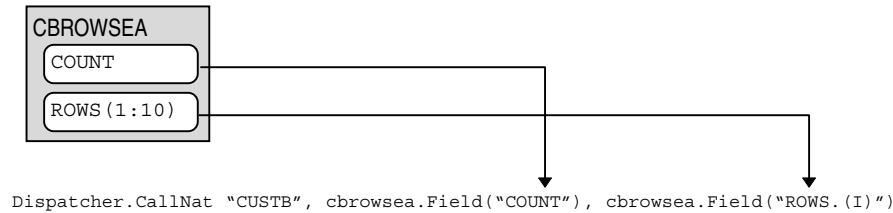
In the previous example, you passed one field and all occurrences of an array into a subprogram. You might think you could do the following with the Spectrum Dispatch client:

Example of how NOT to pass parameters to subprograms

```
Dim cbrowsea As NaturalDataArea

Set cbrowsea = SDCApp.Allocate("CBROWSEA")
cbrowsea.Field("COUNT") = 10
Dispatcher.CallNat "CUSTB", cbrowsea.Field("COUNT"), _
                          cbrowsea.Field("ROWS(*)")
```

In the previous example, the field values within CBROWSEA are passed into the dispatcher object's CallNat method.



CBROWSEA Fields Passed to the CallNat Method

The problem with the previous example is that the Field property returns a value, not an object. The second and subsequent parameters to the CallNat method must be objects of type NaturalDataArea. Because the Field property returns a value (not an object), the CallNat method raises a runtime parameter type mismatch error.

Note: The reason that CallNat method accepts only objects as parameters is so that the dispatcher can maintain references to the objects and update them when the response comes back from the server.

A better way to simulate Natural code is to create separate NaturalDataArea objects for each of the parameters you are sending to the subprogram. The bold text in the example below illustrates these differences.

Example of creating separate NaturalDataArea objects of each parameter

```

Dim cbrowsea As NaturalDataArea
Dim mycount As NaturalDataArea
Dim myrows As NaturalDataArea

Set cbrowsea = SDCApp.Allocate("CBROWSEA")
Set mycount = SDCApp.Allocate("CBA-C")      ' 01 COUNT(N3)
Set myrows = SDCApp.Allocate("CBA-R")      ' 01 ROWS(A32/1:10)

cbrowsea.Field("COUNT") = 10

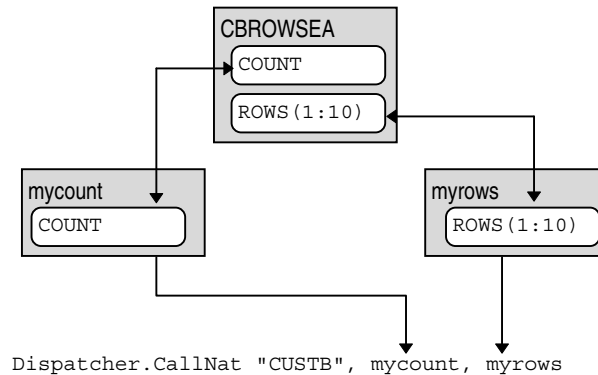
' Copy the CBROWSEA fields into the temporary data areas.
mycount.Field("COUNT") = cbrowsea.Field("COUNT")
myrows.Field("ROWS") = cbrowsea.Field("ROWS")

Dispatcher.CallNat "CUSTB", mycount, myrows

' Copy the fields from the temporary data areas back into CBROWSEA.
cbrowsea.Field("COUNT") = mycount.Field("COUNT")
cbrowsea.Field("ROWS") = myrows.Field("ROWS")

```

In the previous example, the two newly-defined objects (mycount and myrows) are NaturalDataArea objects which contain copies of COUNT and ROWS respectively. These two objects are then passed into the CallNat method.



CBROWSEA Fields Defined as Objects to the CallNat Method

The previous example, showed how to create additional data areas for the individual fields to be passed to the subprogram. These data areas must be initialized from the CBROWSEA data area before issuing the CALLNAT. After the CALLNAT, the data areas must be copied back into CBROWSEA. This code looks quite different than the original Natural code.

A better solution is to create a pointer to a field within a NaturalDataArea object and pass that pointer to the CallNat method of the dispatcher object, effectively passing the field by reference. Construct Spectrum provides the FieldRef property to do just that. This property of the NaturalDataArea class creates an instance of the NaturalDataArea that does not contain its own data, but rather points to a field in the data area that created it.

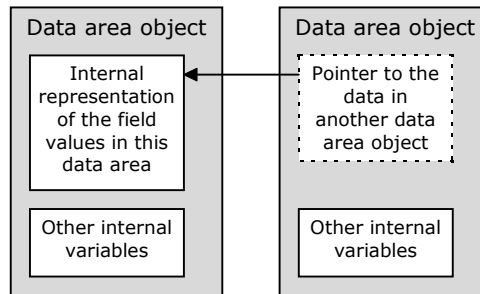
Syntax of the FieldRef property

```
Function FieldRef (ByVal FieldName As String) As NaturalDataArea
```

where:

FieldName	is the field the FieldRef points to
-----------	-------------------------------------

The FieldRef property creates a new instance of NaturalDataArea with the same field definitions as in the field indicated by the FieldName parameter. However, any time a field in this new data is read or written, the field in the original data area is accessed. This effectively creates two data areas that refer to the same data.



Using the FieldRef Property to Create Two Data Areas that Refer to the Same Data

You can now rewrite your original Visual Basic code using the FieldRef property instead of the Field property.

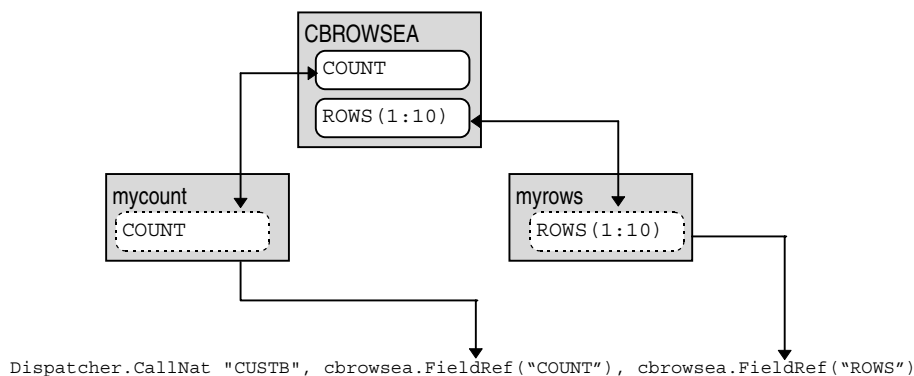
Example of using the FieldRef property

```

Dim cbrowsea As NaturalDataArea

Set cbrowsea = SDCApp.Allocate("CBROWSEA")
cbrowsea.Field("COUNT") = 10
Dispatcher.CallNat "CUSTB", cbrowsea.FieldRef("COUNT"), _
                        cbrowsea.FieldRef("ROWS")
  
```

In the previous example, the FieldRef property creates a temporary Natural-DataArea object that is passed to the CallNat method. Each temporary data area contains a pointer to the original data.



Using the FieldRef Property to Create Two Data Areas that Refer to the Same Data

For a different example, assume the following Natural data area called CUSTA:

Example of using the CUSTA Natural data area

```
01 CUSTOMER-NUMBER (N5)
01 FIRST-NAME (A20)
01 LAST-NAME (A20)
01 MAILING-ADDRESS
  02 STREET (A30)
  02 CITY (A20)
  02 PROVINCE (A20)
  02 POSTAL-CODE (A6)
01 SHIPPING-ADDRESS
  02 STREET (A30)
  02 CITY (A20)
  02 PROVINCE (A20)
  02 POSTAL-CODE (A6)
```

You could use the FieldRef property to obtain a pointer to the Mailing-Address or Shipping-Address structures so that you can process them individually, as the following code sample shows:

Example of Visual Basic Code

```

Dim mycust As NaturalDataArea
Dim mycustref As NaturalDataArea

Set mycust = SDCApp.Allocate("CUSTA")

For i = 1 To 2
    If i = 1 Then
        Set mycustref = mycust.FieldRef("MAILING-ADDRESS")
    Else
        Set mycustref = mycust.FieldRef("SHIPPING-ADDRESS")
    End If

    ' At this point, mycustref is an alias to either the mailing
    ' address or the shipping address fields of the mycust data area.

    With mycustref
        Print .Field("STREET")
        Print .Field("CITY")
        Print .Field("PROVINCE")
        Print .Field("POSTAL-CODE")
    End With
Next

```

The previous example did not specify the level 1 structure name to qualify the field name when reading the Street, City, Province, or Postal-Code fields. This is because the NaturalDataArea object returned by the FieldRef property only contains definitions for the Mailing-Address or Shipping-Address fields.

1:V Fields

In a Natural parameter data area, you may specify an array with a variable number of occurrences by using the index notation 1:V.

Example of specifying an array

```

01 #ROWS(1:V)
02 ...

```

Some of the library image files may already contain similar data area definitions. However, you must specify the number of occurrences for each V to create an instance of this data area. Specify the number of occurrences by using the optional VSubstitutions parameter when you call the Application.Allocate method.

Example of specifying the number of occurrences for your array

```
Function Allocate (ByVal DataAreaName As String, _
                  ParamArray VSubstitutions() As Variant) _
                  As NaturalDataArea
```

For your arrays to operate successfully, you must provide a value for each V in the data area definition or a runtime error will occur. The parameters following DataAreaName in the Allocate call are called a V substitution list.

The following example shows what a call to Allocate would look like.

Example of a PDA

```
[TESTPDA]
01 PARM1 (A5)
01 PARM2 (A3/1:V,1:V)
01 PARM3 (1:V)
   02 PARM4 (N3/1:V)
```

Example of instantiating the PDA

```
[TESTPDA]
01 PARM1 (A5)
01 PARM2 (A3/1:10,1:5)
01 PARM3 (1:20)
   02 PARM4 (N3/1:5)
```

Example of calling the Allocate method

```
Set nda = SDCApp.Allocate("TESTPDA", _
                          "PARM2", 10, 5, _
                          "PARM3", 20, _
                          "PARM4", 5)
```

In the previous example, the V substitution list consists of groups of parameters. Each group identifies a field and provides the substitution values for the 1:V specifications for that field. There must be as many groups as there are fields with 1:V specifications.

You can also store the V substitution list in an array and pass the array as a parameter to the Allocate method.

Example of passing the array to the Allocate method

```
Dim vlist(1 To 7) As Variant

vlist(1) = "PARM2": vlist(2) = 10: vlist(3) = 5
vlist(4) = "PARM3": vlist(5) = 20
vlist(6) = "PARM4": vlist(7) = 5

Set nda = SDCApp.Allocate("TESTPDA", vlist)
```

When you read the FieldDef property to obtain the lower and upper bounds of an array that was defined with 1:V, 1 will be returned for the lower bound and the value specified for that field's V will be returned for the upper bound.

Example of obtaining the upper and lower bounds of an array

```
With nda.FieldDef("PARM4")
    Print .FromIndex(1) & ":" & .ThruIndex(1)      ' Prints "1:20"
    Print .FromIndex(2) & ":" & .ThruIndex(2)      ' Prints "1:5"
End With
```

You can create instances of the same data area with different numbers of occurrences.

Example of using the same data area with varying numbers of occurrences

```
Dim data1 As NaturalDataArea
Dim data2 As NaturalDataArea

Set data1 = SDCApp.Allocate("#ROWS", 15)
Set data2 = SDCApp.Allocate("#ROWS", 100)

With data1.FieldDef("#ROWS")
    Print .FromIndex(1) & ":" & .ThruIndex(1)      ' Prints "1:15"
End With

With data2.FieldDef("#ROWS")
    Print .FromIndex(1) & ":" & .ThruIndex(1)      ' Prints "1:100"
End With
```

Note: The total size of the data area must be no greater than that allowed by Natural.

CREATING SPECTRUM APPLICATIONS WITHOUT THE CLIENT FRAMEWORK

This chapter describes the process of creating a Construct Spectrum application without using Construct-generated components. You will learn how to create and deploy your application by working through the steps of creating a simple application. While this simple application is designed to run using Microsoft Visual Basic, you may choose to use any other development tool that fully supports OLE automation.

The following topics are covered:

- **Setting Up the Server Components**, page 298
- **Generating Subprogram Proxies Using the Spectrum Models**, page 303
- **Creating the Library Image Files (LIFs)**, page 307
- **Developing the Client Application**, page 311

For more information about creating Construct Spectrum applications using components that have been generated using earlier versions of Natural Construct, see **Moving Existing Applications to Construct Spectrum**, page 245, *Developing Client/Server Applications*.

Setting Up the Server Components

This section describes the process of setting up server-side components to prepare an environment for the client to be able to communicate with the server. You can create server-side components entirely within the Natural environment.

It includes the following sections:

- Creating or selecting application services
- Example of creating a simple Natural subprogram

Creating or Selecting Application Services

When creating new application services or selecting application services for deployment in a client/server environment, ensure that the Natural subprograms follow certain rules. Natural subprograms primarily execute as remote services in environments where no input and output devices are defined. Therefore, there are some restrictions imposed on your Natural programs.

The following sections identify issues to consider when you are developing new application services or adapting existing services to execute in a client/server environment.

No Terminal I/O

Avoid the use of all commands that require input from the user or write information to any external source other than a database file. This includes the INPUT statement as well as the WRITE, PRINT, and DISPLAY statements.

While the INPUT statement cannot be used to input data from the user, you can use the INPUT statement to retrieve data that has been stacked using the STACK TOP DATA statement.

The WRITE, PRINT, and DISPLAY statements should only be used to write information to the Natural source area for the purpose of debugging your application. If, however, you are running servers in batch mode, you can send the WRITE, PRINT, and DISPLAY statements to the batch output queue. This data is only viewable after the batch job ends.

For more information, see **Debugging Your Client/Server Application**, page 201.

Subprogram Interface

Construct Spectrum is only able to communicate with application services that are implemented as subprograms. If necessary, you may invoke programs from inside the called subprograms by using the `FETCH RETURN` statement.

No Global Data Area (GDA)

Normally, called services do not define a Global Data Area (GDA) because the contents of GDAs used by a subprogram are not preserved between calls. However, when necessary, GDAs can be used as a means to overcome a shortage of local data area (LDA) storage.

Parameter Data Area (PDA) Data Size Limitation

All data transmitted between the client and the server is converted into printable characters. For example, an I2 integer requires 6 bytes of data during transmission: a sign byte and five digits. The size of this converted data cannot exceed 32K. When you check or catalog a subprogram proxy, Natural will display an error if the size of the converted data exceeds 32K.

Subprogram Behavior

All subprograms to be invoked as application services must return to the calling routine. This requires that the subprogram and any called routines do not execute statements such as `STOP`, `FETCH`, or `TERMINATE`. They should also avoid statements that will affect the caller, such as `RELEASE STACK`, `STACK COMMAND`, `STACK DATA`, and `RELEASE VARIABLES`. Finally, called subprograms should not modify the `*ERROR-TA` value.

Externalize Parameters

The best design strategy is to externally define the parameters of your subprogram proxies. Only when the parameters are externally defined can the Parameter Data Areas be downloaded and incorporated into the Spectrum Dispatch Client.

However, Construct Spectrum does allow you to generate subprogram proxies for subprograms that define their parameters inline. If the subprogram accepts or returns large amounts of data which is strictly input parameters or output parameters, consider grouping all of the input parameters into one level 1 structure and all of the output parameters into another level 1 structure. Parameters that are both input and output should be grouped into a third level 1 structure. This allows the data being sent between the client and the server to be optimized such that only input data is sent to the server and only output data is returned to the client.

Timing Issues

Some application services perform tasks that require extensive processing. The length of time spent in the application service affects the timeout values triggered in a client system. If an application requires extensive time to execute, it may be necessary to define and associate such long-running processes with Spectrum dispatch services that use inflated timeout values. These resource-intensive application services can be made to execute under a specially-configured dispatch service. This includes defining special services in the EntireX Broker attribute files.

Example of Creating a Simple Natural Subprogram

The next step describes how to create a small application service and generate the associated subprogram proxy.

- To create your server-based components for the sample application:
- 1 Create a parameter data area named GCDA with the following configuration and compile it in the SAMPLE library.

```

Parameter GCDA      Library SAMPLE                      DBID  17 FNR  38
Command
I T L Name          F Leng Index/Init/EM/Name/Comment    > +
All - -----
    1 GCD-DATA
    2 #OPERAND-1      I    4
    2 #OPERAND-2      I    4
    2 #RESULT         I    4

----- Current Source Size: 143  Free: 43402 ----- S 4    L 1

```

Example of Parameter Data Area GCDA

An example of this module is found in the library SYSSPEC with the name SAMPLE_A.

- 2 Type the following code into the Natural editor and compile it as a Natural subprogram named GCDN in SAMPLE library.

```

>                                     > + Subprogram GCDN      Lib SAMPLE
Top      ....+....1....+....2....+....3....+....4....+....5....+....Mode Struct..
0010 *****
0020 ** This module accepts numbers as input paramters and returns the
0030 ** greatest common divisor as the result.
0040 *****
0050 DEFINE DATA
0060     PARAMETER USING GCDA          /* Input and output parameters
0070     LOCAL
0080     01 #TEMP(I4)                  /* Local variable used in calculation
0090 END-DEFINE
0100 **
0110 ** Repeat while the second operand value is not equal to 0.
0120 REPEAT
0130     WHILE #OPERAND-2 NE 0
0140     DIVIDE #OPERAND-2 INTO #OPERAND-1 REMAINDER #TEMP
0150     ASSIGN #OPERAND-1 = #OPERAND-2
0160     ASSIGN #OPERAND-2 = #TEMP
0170 END-REPEAT
0180 **
0190 ASSIGN #RESULT = #OPERAND-1
0200 END
      ....+..Current Source Size: 799 Char. Free: 42970 ....+... S 21   L 1

```

Example of Natural Subprogram GCDN

An example of this module is found in the SYSSPEC library with the name SAMPLE_N.

Generating Subprogram Proxies Using the Spectrum Models

This section describes the steps required to make the newly-created GCDN Natural subprogram accessible from the client. The Subprogram-Proxy model was specifically designed to automate this process.

This section includes the following topics:

- Using the model
- Application service definition

Using the Model

The Subprogram-Proxy model is available in the Generation subsystem on the server and as a model wizard in the Construct Windows interface. In this example, we show the model wizard.

For information about using the model, see **Using the Subprogram Proxy Model**, page 129. For information about using the Construct Windows interface, see **Using the Construct Windows Interface**, page 77 of the *Natural Construct Generation User's Manual*.

➤ To generate a subprogram proxy for the GCDN subprogram:

- 1 Fill in the text boxes as shown below:

The screenshot shows the 'SUBPROGRAM-PROXY Wizard' dialog box with the 'Standard Parameters' tab selected. The dialog has a sidebar on the left with three buttons: 'Start', 'Standard Parameters' (which is highlighted with a square icon), and 'Finish'. The main area is titled 'Enter the standard parameters for this model'. It contains several input fields: 'Module:' with the value 'GCD', 'System:' with 'DEMO', 'Title:' with 'Subprogram proxy for GCD', and 'Description:' with a text area containing 'This subprogram proxy communicates with the module that calculates the greatest common divisor of two numbers'. Below these are 'Subprogram:' with 'GCDN' and a browse button (...), 'Domain:' with 'SAMPLE' and a browse button (...), 'Object name:' with 'GCDN|', and 'Version:' with '1.1.1'. To the right of the 'Subprogram' and 'Domain' fields is a button labeled 'Edit 1:V Overrides'. At the bottom of the main area are three checkboxes: 'Generate trace code', 'Compress network data', and 'Encrypt network data', all of which are currently unchecked. The bottom of the dialog features five buttons: 'Validate', 'Cancel', '< Back', 'Next >', and 'Finish'.

Start	Enter the standard parameters for this model	
Standard Parameters	Module:	GCD
Finish	System:	DEMO
	Title:	Subprogram proxy for GCD
	Description:	This subprogram proxy communicates with the module that calculates the greatest common divisor of two numbers
	Subprogram:	GCDN ...
	Domain:	SAMPLE ...
	Object name:	GCDN
	Version:	1.1.1
	<input type="checkbox"/> Generate trace code	
	<input type="checkbox"/> Compress network data	
	<input type="checkbox"/> Encrypt network data	
	Validate	Cancel
	< Back	Next >
	Finish	

Subprogram-Proxy Wizard — Standard Parameters

Many of the input values for the Subprogram-Proxy model are automatically determined and set by the model itself.

- 2 Generate and stow the subprogram proxy GCD. Generating the subprogram proxy model results in the creation of two new items:
 - the generated subprogram proxy GCD
 - the application service definition, which is generated into the Spectrum Administration subsystem.

Application Service Definition

The application service definition is automatically created when you generate using the Subprogram-Proxy model. It describes the target subprogram to the Spectrum dispatch service by defining the name and location of the subprogram and which methods it supports.

This target subprogram can be any Natural subprogram. There are many advantages, however, to creating these Natural subprograms using the Object-Maint-Subp and the Object-Browse-Subp models.

If you choose to tailor your application service definition, use the following procedure. This example continues to use the example subprogram proxy GCD.

- To access the generated application service definition:
- 1 Access the Construct Spectrum Administration subsystem main menu and enter “AA” in the Function field.
The Application Administration main menu is displayed.
 - 2 Enter “MM” in the Function field.
The Application Administration Maintenance menu is displayed.
 - 3 Enter “AS” in the Function field.
The Maintain Application Service Definitions panel is displayed.
 - 4 Type “D” for Display in the action field.

- 5 Type “SAMPLE” in the Domain field, “GCD” in the Object field, and “01/01/01” in the Version field and press Enter.
The application service definition associated with the subprogram proxy GCD is displayed.

```

BSIF_MP          Construct Spectrum Administration Subsystem          BSIF_11
June 27          Maintain Application Service Definitions              3:15 PM

Action (A,B,C,D,M,N,P)  _

Domain.....: SAMPLE_ *
Object.....: GCDN
Version.....: 01 / 01 / 01
Description.....: GCDN
Default subprogram proxy: GCD_
Steplib.....: _____ *

01          Method Name          Subprogram          Steplib *
-----
1 DEFAULT
2
3
4
5

Command:
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---
confm help retrn quit          flip pref bkwrdr frwrdr          main
Appl Srvc Definition DEMO-PRODUC displayed successfully

```

Example of the Application Service Definition Panel

The application service definition for the subprogram proxy GCD record was automatically generated with a single method — the Default method. The Default method is generated automatically for each application service definition unless the target Natural subprogram was generated by either the Object-Maintenance-Subp or Object-Browse-Subp model.

This application service definition does not specify a steplib, although one is required to access the target object. This is because the specified domain, SAMPLE, already has a steplib defined. Therefore, this application service definition, by default, also uses the SAMPLE steplib.

For more information on defining steplibs, **Define a Steplib Chain**, page 52.

Creating the Library Image Files (LIFs)

After reading this section, you will understand what library image files (LIFs) are and how to use and tailor your applications' library image files.

Before you can call the subprogram from the client application, you must create a file on the client that describes the subprogram and any parameter data areas it uses. This file is called a "library image file" because it contains an image (or a copy) of the Natural objects from your application library on the server platform. Definitions of all objects you will be using in the client application must be in the library image file. If your client application uses objects on the server platform from multiple libraries, you need to create one library image file for each library.

The filename for a library image file is the same as the name of the library, plus the extension .LIF. For example, if you have an application library called CST-DEMO, the library image file on your client will be called CSTDEMO.LIF. All library image files the client application uses must be stored in the same LIF directory on your PC (or on a network file server).

Each application may have its own LIF directory, or multiple applications can share a single LIF directory that contains many different library image files.

Use the Construct Spectrum Add-In in Visual Basic to create the library image file. The download function in the Construct Spectrum Add-In downloads definitions from an application library on the server to your client.

Using the Construct Spectrum Add-In

Use the Construct Spectrum Add-In to download the subprogram definition and parameter data area (PDA) definitions into a library image file.

Before You Start

- Ensure that you know the library name, the database ID (DBID), and the file number (FNR) of the FUSER system file. This file resides on the server and contains the subprogram you created earlier in an application library on a FUSER file.
- Choose or create a LIF directory on your PC for the library image file.

Note: You will also create the client application in this directory in a subsequent step.

- Ensure that you know the name of the subprogram proxy and all parameter data areas used by the subprogram.
- Check that the Spectrum Dispatch Client has been installed and configured properly. For more information, see the *Construct Spectrum Client Installation Guide*.

Downloading Definitions

- 1 Start Visual Basic, if it is not already running.
- 2 From the Visual Basic **Add-Ins** menu, select **Construct Spectrum**, then select **Download Generated Modules**.

The **Download Modules** dialog box appears:

Name	Type	Model	User ID	Date / Time

Resulting Download Dialog

- 3 Enter the name of the application library, the database ID (DBID), and the file number (FNR) of the FUSER file that contains the library.

Note: If you have used the download function before, the DBID, FNR and the library name, that you used last time will be filled in automatically.

- 4 Enter a wildcard pattern such as “*” in the **Module name** text box to list all of the modules in the library, or enter a more specific search pattern to narrow the list.

Name	Type	Model	User ID	Date / Time
------	------	-------	---------	-------------

Searching in the Module Field on the Download Dialog

- 5 Press Enter or click **List**.

After a few seconds, a list of the modules matching the wildcard pattern will be displayed. If an error message is displayed, see **Downloading the Client Modules**, page 229 in *Developing Client / Server Applications* or *Construct Spectrum Messages*.

Note: You will only see subprogram proxies and parameter data areas in this list. Other Natural object types such as programs, maps, and copy code members will not be displayed because they cannot be downloaded.

- 6 Select all of the parameter data areas (PDAs) and the subprogram proxies associated with your subprogram.

Tip: To select more than one item in the list, use the standard Windows multiple-select actions (Shift-Click and Ctrl-Click), or use the mouse to drag a marquee around the items you wish to select.

- 7 Press Enter or click **Download**.
- 8 After all of the selected modules have been downloaded, click **Close** on the **Download Modules** dialog box.

If an error message is displayed during the download process, see *Construct Spectrum Reference Manual*.

The download process creates a new library image file in the LIF directory or updates an existing LIF. In the next section, you will see how to develop a client application that uses the definitions in the LIF to call the subprogram on the server platform.

Developing the Client Application

This section describes the minimum requirements to develop a client application that calls your subprogram. Although this example uses Microsoft Visual Basic Version to develop the application, you may choose any other development tool that supports OLE automation.

This section assumes you are familiar with the following OLE automation concepts. If any of these are unfamiliar, refer to the appropriate documentation for the development tool you are using.

OLE Automation Term	Definition
object library (or type library)	Provides definitions of all the objects, methods, and properties an OLE automation server exposes
externally creatable object	Object exposed by an OLE automation server that can be created from outside the server
dependent object	Object exposed by an OLE automation server that can be accessed only by using a method of a higher-level object, such as an externally-creatable object

There are six main steps to follow when developing your client application:

Step	Description
Step 1	Create a project
Step 2	Add a reference to the object library
Step 3	Write code to initialize the Spectrum Dispatch Client
Step 4	Create the user interface
Step 5	Write the code that does the call
Step 6	Run the application

Each of these steps is described in the sections which follow.

Step 1 — Create a New Project

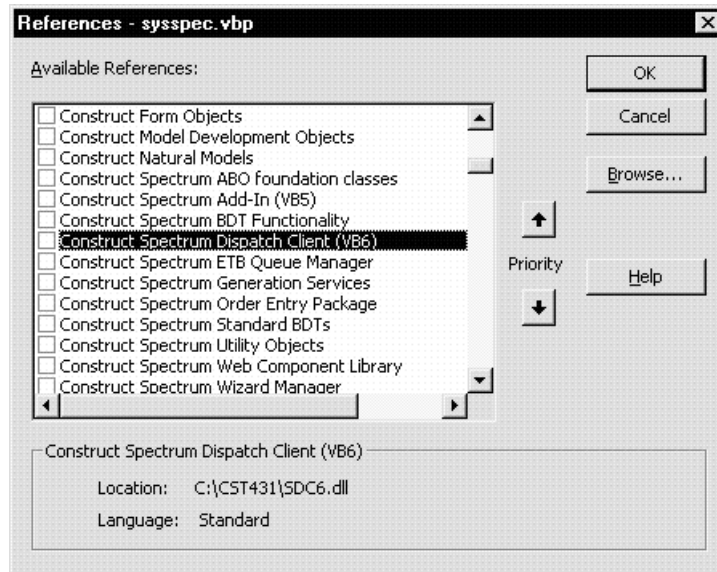
Start Visual Basic and create a new project. Save all project components in the LIF directory you created earlier. This makes keeping track of all project components easier.

- To create a new project:
 - 1 Start Visual Basic.
 - 2 Create a new Standard EXE project.

Step 2 — Add a Reference to the Object Library

Before you can use the objects in the Spectrum Dispatch Client, add a reference to its object library in your Visual Basic project.

- To add a reference to the Spectrum Dispatch Client's Object library:
 - 1 From the **Tools** menu, click **References**.
 - 2 Select **Construct Spectrum Dispatch Client** (VB6) in the **References** dialog box by ensuring it is checked.



Using the References Dialog to Select the Spectrum Dispatch Client

- To use Visual Basic's **Object Browser** dialog box to view the object library:
 - 1 From the **View** menu, click **Object Browser** or press F2.
 - 2 Select **SDCLib** in the **Libraries/Projects** box on the **Object Browser** dialog box.

Step 3 — Write Code to Initialize the Spectrum Dispatch Client

- To write code to initialize the Spectrum Dispatch Client:
 - 1 From the **Project** menu, click **Add Module** to add a new module to your Construct Spectrum project.
 - 2 Add the following code to the module:

```
Public SDCApp As New SDCLib6.Application
Public Dispatcher As SDCLib6.Dispatcher

Public Sub Main
    SDCApp.Initialize App.Path, "CSTDemo"
    SDCApp.UserID = "GUEST"

    Set Dispatcher = SDCApp.CreateDispatcher()
    Dispatcher.DisplayErrors = True

    Form1.Show
End Sub
```

where:

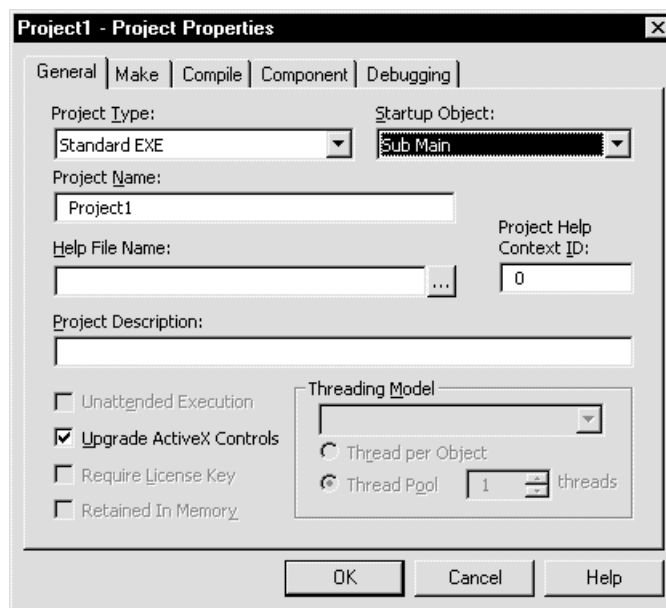
Application	is an externally-creatable object exposed by the Spectrum Dispatch Client. It is used to create all other objects.
App.Path	is a Visual Basic property that returns the name of the directory that contains your saved project. In this example, the project is stored in the LIF directory. App.Path returns the name of the directory where your executable project is located. Your library image file must always be in that directory.
Dispatcher	is an object used to communicate with the server platform
Form1	contains the user interface for the client application
Initialize method	Tells the Spectrum Dispatch Client the name of the LIF directory and the name of the application library. Together, these two values tell the Spectrum Dispatch Client the name of the library image file.

where: (continued)

Set Dispatcher = CreateDispatcher()	Creates a dispatcher object with methods that allow you to call the subprogram on the server platform. Setting the DisplayErrors property to True causes the dispatcher object to automatically display any communication errors. You will not have to write additional code to display errors.
"GUEST"	Is a predefined user ID in the Spectrum Administration Subsystem that contains the required security definitions for this example. Every call to the server platform requires the caller's user ID to be known.

This code creates two object variables that you will use throughout the application: SDCApp and Dispatcher.

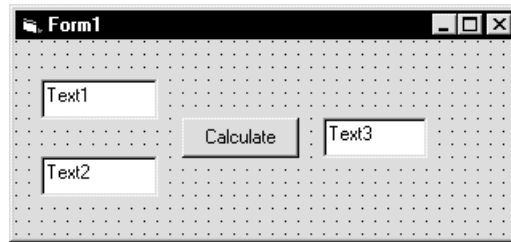
- 3 From the **Project** menu, click **<App.Name> Properties**.
- 4 Set the **Startup Object** to "Sub Main" on the **General** tab.



Project Properties Window

Step 4 — Create the User Interface

- To create the user interface for this client application:
 - 1 Ensure Form1 is open in design mode.
 - 2 Add three TextBox controls and a CommandButton control, arranged as in the following illustration:



Example of the Layout of Form 1

- 3 Set the control properties as follows:

Control	Property	Value
Text box 1	(Name)	txtOperand1
	Text	<empty>
Text box 2	Type	txtOperand2
	Text	<empty>
Text box 3	Type	txtResult
	Text	<empty>
CommandButton	Type	cmdCalculate
	Caption	Calculate

Step 5 — Write the Code that Does the Call

- To add code to the Click event of the command button which will call the subprogram:
 - 1 Double-click the command button to open the code window.
 - 2 Add the following code to the cmdCalculate_Click procedure:

```

Private Sub cmdCalculate_Click()

    Dim parms As NaturalDataArea

    Set parms = SDCApp.Allocate("GCDA")

    parms("#OPERAND-1") = Val(txtOperand1.Text)
    parms("#OPERAND-2") = Val(txtOperand2.Text)

    Screen.MousePointer = vbHourglass

    Dispatcher.CallNat "GCDN", parms

    Screen.MousePointer = vbDefault

    If Dispatcher.Successful Then
        txtResult.Text = parms("#RESULT")
    End If
End Sub

```

This code declares and allocates a Natural data area that corresponds to the parameter data area expected by your subprogram. Next, it assigns the numeric values in the two text boxes, txtOperand1 and txtOperand2, to the #OPERAND-1 and #OPERAND-2 fields in the data area. It then calls the subprogram with the CallNat method of the dispatcher object. The mouse pointer changes to an hourglass icon for the duration of this call. Finally, if the call was successful, the contents of the Result field are displayed in the txtResult textbox. If the call was unsuccessful, the dispatcher object automatically displays the error message because you set the DisplayErrors property to True.

Step 6 — Run the Application

Before running the application, save the project. This ensures that the App.Path property in Sub Main returns the correct directory for the Initialize call.

Tip: If you forget to save after creating a new project, App.Path returns the working directory from which you started Visual Basic. However, when you save on disk, App.Path returns the name of the directory in which the project file is saved.

- To use the application:
- 1 Run the application by pressing F5.
 - 2 Enter a number into each operand textbox.
 - 3 Click **Calculate**.
The result appears in the **Result** textbox.

Note: It is common for the first call to the communication server platform to take a few seconds. This is because the EntireX Broker DLLs must be loaded into memory and initialized. Subsequent calls to the server platform will be much faster.

- 4 Close the window to return to Visual Basic's design mode when you are finished testing the application.

The dispatcher object may display an error while you are testing the application. One error that you may receive in the cmdCalculate_Click procedure is the Numeric Overflow error:

Error	Possible Causes	Resolution
Numeric overflow	The value being assigned to the #OPERAND-1 or #OPERAND-2 field is too large for the Natural format. For example, if you assign 532 to a field of format N2.	Enhance the code to check that the values entered by the user into the textboxes are not too large for the Natural format.

If the dispatcher object displays an error while you are testing the application, see **Debugging Your Client/Server Application**, page 201 or *Construct Spectrum Messages* for more information.

See **Deploying Your Client/Server Application**, page 237 for information on the procedure required to package the client application and install it onto another PC.

APPENDIX A: GLOSSARY

The following terms are used throughout the Construct Spectrum documentation set. Each term is listed with its meaning:

Term	Definition
ActiveX business object (ABO)	A Visual Basic class used as a proxy on the client for a Natural business object. The ABO wraps the Spectrum calls required to communicate with a Natural subprogram exposed by a subprogram proxy.
active DLL	A DLL containing one or more ActiveX business objects. Used to package and deploy web applications.
application library	Natural library containing the server application components of a client/server application.
application service	Natural subprogram that implements methods that can be called as remote services.
application service definition	Definition within the Construct Spectrum Administration subsystem that identifies the methods exposed by a subprogram. This definition is created automatically by the Subprogram-Proxy model. These settings can be modified using the Maintain Application Service Definition panel in the Construct Spectrum Administration subsystem.
architecture	A high-level description of the organization of functional responsibilities within a system. The goals of an architecture are to convey some information about the general structure of systems. An architecture defines the relationship of system components, but does nothing to describe the specific implementation of those components. Client/server is an architecture because it defines a relationship between system components.

Term	Definition (continued)
browse command handler	Defines commands that are linked to a browse dialog. Also acts as the initial target of these commands, typically redirecting the commands to other application components. See also command handler , page 324.
browse data cache	Area that contains database records returned from the server. These are usually displayed on a browse dialog.
browse dialog	Generic GUI browse dialog called to display any browse data including that residing on a mainframe or PC.
browse process	<p>Process by which framework components and generated browse components retrieve data and, optionally, display it in a browse dialog.</p> <p>For example, a browse process can be used to retrieve rows of data, examine specific values, perform calculations on values, and perform conditional processing. Users can choose to display the results in a browse dialog, but this is not required.</p>
business data type (BDT)	<p>A type validation on the client that applies business semantics to a field. Typically, BDTs are used to format field data specified by the user.</p> <p>For example, if an application has a field for entering a phone number, you can associate a BDT with the field to reformat the phone number with hyphens. The user might enter "7053332112". When the user moves to the next field or performs some other action, the number is automatically reformatted as 705-333-2112. Construct Spectrum comes with standard BDTs that can be customized. Users can also create their own BDTs. BDT modifiers are added to the keyword components of a field defined in Predict.</p>
business data type class	Collection of all business data type procedures.
business data type controller class	Collection of methods available to members of a class.

Term	Definition (continued)
business data type controller object	A component of the Construct Spectrum client framework. Each application declares a business data type controller object (BDT controller). Each BDT controller is an instance of the business data type controller class, and uses the methods available to that class. Among its methods, the BDT controller records and maintains a list of names for each business data type used in addition to a pointer to the business data type definition. See also business data type (BDT) , page 322.
business data type modifier	Additional logic users supply to modify the formatting or validation rules for a business data type. For example, the business data type <code>BTD_NUMERIC</code> ensures that only numeric values are entered in a field. Users can add a modifier to also specify that the numeric values will be rounded. Each business data type defines its own set of modifiers to increase its flexibility.
business data type procedure	Code that implements a business data type.
business object	Conceptual abstraction that groups together all the attributes and behaviors associated with a business entity, for example, Customer or Order. See also Visual Basic business object , page 339.
BUSINESS-OBJECT-SUPER-MODEL	A model (available in the Construct Windows interface and Generation subsystem) that generates multiple modules for both web or client/server applications that do not use the Construct Spectrum client framework.
Business- object super model wizard	Wizard that generates maintenance and browse subprograms and subprogram proxies for business objects.
cardinality	Indicates the number of information dimensions. Information with the same number of dimensions has the same cardinality.
child model	An individual model for which a Super model (parent model) collects parameters and generates specifications.

Term	Definition (continued)
client application	The portion of a Construct Spectrum client/server application that runs on a Windows platform.
client framework	A set of cooperating classes forming a reusable design that are supplied with a Construct Spectrum project in Visual Basic. The client framework provides a skeleton of functionality that can be either customized or filled with generated and hand-coded Visual Basic modules. The client framework includes forms, classes, procedures, global variables, and constants that are shared among generated application components. The client framework both reduces the size of the generated components and allows them to interact. The Construct Spectrum framework includes both client and server components that can be customized to meet user needs.
code block	One or more lines of code in a Visual Basic code module. These lines can be manipulated in the code editor as a block.
command block	Block of code that requires the Natural Construct nucleus to treat the text within the block as a separate module. Also specifies that the nucleus apply the specified command to this block of text. Command blocks are used by super models generating multiple modules.
command handler	An object, generally a Visual Basic class, that processes a command. Command handlers are called by the client framework when a user clicks a menu command or a toolbar button. A command handler may handle multiple commands. See also command handler list , page 324 and hook , page 329.
command handler list	List of command handlers for each command ID. The last command handler to be hooked to a command ID is called “first”. See also hook , page 329.

Term	Definition (continued)
command IDs	Unique identifiers for application-specific commands that menus and toolbar buttons send. Define these commands by specifying a single command ID as “constant” for each unique menu and toolbar command.
complex redefine	Involves the redefinition of a data area containing multiple data types, multiple redefinitions of a data field, or multiple levels of field redefinitions.
compression	Reduces the total byte size required for a given set of data. Construct Spectrum lets users compress data transmitted both to and from the client to the server. In both cases, the data is compressed when it is sent and then decompressed when it reaches the destination. This technique reduces the size of data transmissions and improves the performance of a network.
Construct Spectrum	A Construct Spectrum application consists of both a client and server component. The client component is a Construct Spectrum application running in Visual Basic. The server component is a set of subprograms accessed remotely by the client component.
Construct Spectrum Add-In	Customized functionality added to the Visual Basic environment.
Construct Spectrum Administration subsystem	Mainframe subsystem used to maintain and query the tables that define Construct Spectrum application services and security.
database record	A logical view of database information. A database record can be comprised of one or more database files or tables that are logically related. In Construct Spectrum, database information is represented in parameter data areas (PDAs).
DBID	Acronym for database ID, which is the number identifying the server database that contains application components.

Term	Definition (continued)
debugging tools	Construct Spectrum lets users simulate client calls online so they can use the traditional debugging tools that locate and analyze logic errors. These debugging tools include trace options, input and output statements, and the Natural Debugging facility. Trace options let users save data from a client call to a file on the server. In online simulations, users can use the data in this file to recreate situations that caused errors. Adding INPUT, PRINT, and WRITE statements to code lets you step through the program to see how the execution works. Online simulations also lets users use the Natural Debugging facility to establish a debug environment. See Natural Debugging Facility in the <i>Natural Utilities Manual</i> .
dependent object	An object exposed by an OLE automation server that can only be created using the method of a higher-level object. See also externally-creatable object , page 327.
deployment	Movement of an application from a development environment to a production environment.
dialog	GUI form running on the client.
dispatch service	A server component used to broker communications between server application components and client framework components. See also Spectrum dispatch service , page 336.
domain	An entity used to define a collection of related business objects (for example, Test, Admin, and Sales).
double byte character sets (DBCS)	Used to represent characters in some non-Latin languages.
download data	The transfer of modules from the server to the client.

Term	Definition (continued)
encapsulation	A technique in object-oriented programming in which the internal implementation details of an object, for example, a customer order object, are hidden from the users of the object. The object methods control how the object's data is manipulated. Encapsulation is important because it allows internal implementations to change over time without affecting the way an object is used externally.
encryption	Encoding data so it is unusable for individuals without access to the necessary decryption algorithms. Construct Spectrum facilitates the encryption of sensitive data, for example, payroll information during network transmission. Data is decrypted when it reaches its destination.
Entire Broker service settings	A collection of Entire Broker-related parameters including Entire Broker ID, server class, server name, and service.
Entire Broker stub	The Entire Broker DLL on Windows.
event	An action recognized by an object, for example, pressing a key or clicking a mouse. Developers write code to respond to events.
externally-creatable object	An object exposed by an OLE automation server that can be created from outside the server. See also dependent object , page 326.
field	Component of a database record. The term also refers to areas on a panel in which values are entered.
FNR	Acronym for the file number that identifies a specific server database file containing application components.
foreign key	A key field pointing to a record in an external file. For example, the Construct Spectrum demo application has an ORDER file that has a foreign key to the Warehouse field residing in the WAREHOUSE file. Foreign keys can be set up with a browse function, enabling users to search for and select values.

Term	Definition (continued)
form	<p>Window that acts as the interface for an application. Developers add controls and graphics to a form to create the effect they want. Construct Spectrum supplies forms as part of the client framework in Visual Basic. Construct Spectrum also generates form modules for business object maintenance dialogs.</p> <p>When developers run the project, forms are compiled into GUI dialogs that the user can interact with while using the application. Some forms, including the generic BrowseDialog form, are dynamically configured at run-time by the client framework to alter the look of the form.</p> <p>Form definitions are saved in files with the extension .frm.</p>
form section	<p>A portion of a web page that contains a block of related information.</p>
framework templates	<p>Provide structure/containers for applications. These customizable templates include header, footer, navigation, messages and constants.</p>
generate	<p>The process of creating code from specifications.</p>
generated module	<p>A generated component for either the client or server portion of an application. Examples of generated server modules include a Natural subprogram, subprogram proxy, and parameter data area. Examples of generated client modules include an object factory, dialog, and maintenance object.</p>
generation data cache	<p>An in-memory hierarchical data structure that allows developers to quickly retrieve stored generation data.</p>

Term	Definition (continued)
grid	Allows 2-, 3-, and 4-dimensional information to be displayed in a client/server application. Zero-dimensional (ascalar) data can only show one type of data, such as a phone number, name, or quantity. Most fields are zero-dimensional. Two-dimensional data can show additional information in a grid or table. For example, the detail lines on a customer order can be shown in a grid with each grid row corresponding to a unique line item. Each column in the grid corresponds to a discrete piece of information about the line, such as an item name, price, or quantity.
grid control	GUI control that displays related information in a table format. For example, purchase order line items can be displayed in a grid. The grid control supplied with Construct Spectrum is highly configurable and, by default, sizes itself to the minimal width required to display all the grid components.
group	A collection of users defined in the Construct Spectrum Administration subsystem.
group project***8	
GUI	Acronym for graphical user interface.
GUI control override	Use of Predict keywords to force a particular GUI control derivation.
hook	Associates a command handler object with a command ID. See also command handler , page 324, command handler list , page 324, and command IDs , page 325.
host	See server , page 335.
HTML fragment	A snippet of HTML, not a complete web page.
HTML template	HTML that may contain replacement tag which are dynamically exchanged for content or nested HTML templates.

Term	Definition (continued)
HTML template wizard	A wizard used to generate HTML templates.
http request	A parameterized list of named pair values sent by a browser client to a web application.
instantiation	Process of creating an instance of a class. The result is an object.
internationalization	Adapting an application to make it easy to localize. See also localization , page 330.
job control language (JCL)	Command language used in batch jobs to tell the computer what to do.
keyword	Predict metadata type that acts as a label or identifier. Keywords can be linked to others. Predict metadata types include files, fields, and relationships.
Level 1 data block	A level one field or a level one structure with its subfields in a Natural parameter data area (PDA).
Level 1 data block optimization	A technique to improve the performance of client/server applications by reducing the volume of data transmitted across a network. Rather than sending all data blocks associated with an application object, only those data blocks required are sent.
library image file (LIF)	File used to define Natural definitions used by the Construct Spectrum Dispatch Client.
LIF definitions module	The .bas module in a Visual Basic project containing the definitions for application services, parameter data areas, and subprograms.
localization	The process of translating and adapting a software product for use in a different language or country.

Term	Definition (continued)
lookups	A browse is a type of lookup. Other lookups return descriptive information when users enter a value in a foreign key field on a maintenance dialog. For example, suppose that a Warehouse Number field is a foreign key field on the Order dialog and a descriptive field, the Warehouse Name field, has been attached to the foreign key value. When a user enters a valid warehouse number, the lookup returns the name of the warehouse so that it can be displayed beside the warehouse number on the dialog.
maintenance dialog	A GUI dialog from which a user can perform one or more actions on a business object. For example, a Customer Order object can be represented on a maintenance dialog. Using this dialog, a user can add, delete, or update customer order information if permitted.
MDI child	In a client/server application, any window or dialog displayed in an MDI parent window. Child windows or dialogs are opened from a parent window. For example, the Order maintenance dialog in the demo application is an MDI child to the MDI frame window.
MDI frame	A standard Visual Basic MDI frame that is supplied with the Construct Spectrum client framework.
MDI parent	In a client/server application, an MDI window from which other windows are opened and displayed. The MDI frame supplied with the client framework is an MDI parent.
menu	<p>On a mainframe server, a panel or window listing available functions. To access a function, users type a value in an input field or move the cursor to the value and press Enter.</p> <p>In Windows, a pull-down dialog listing the available functions. To access a function, users select an option from the menu using the cursor or a keystroke combination.</p>

Term	Definition (continued)
menu bar	Displays the menus users select. By default, Construct Spectrum client/server applications contain File, Edit, Actions, Window, and Help menus, each containing standard menu commands.
metadata	Information about data. Metadata describes how physical data is formatted and interrelated. It includes descriptions of data elements, data files, and relationships between data entities. Typically, metadata is maintained in a repository known as a data dictionary, such as Predict.
method	A procedure that operates on an object and is implemented internally by that object. For example, the Update method could be used to update a Customer Order object with any changes to the order information.
model	Template used to generate modules. Each model contains one or more specification panels. Using these panels, developers specify the parameters of a desired module and then generate the corresponding code. Natural Construct provides numerous models, including the Object-Maint-Subp model and the Subprogram-Proxy model wizard.
module	A single application component. A module can be, for example, a hand coded Natural program, subprogram, or data area or a Natural Construct-generated program, subprogram, data area, subprogram proxy.
multi-level security	Security that can be defined at a high level or at a detailed level affecting many objects. For example, users can apply multi-level security to domains, objects, and methods.
multiple-document interface (MDI)	A Microsoft Windows paradigm for presenting windows whereby a parent window can encompass one or more child windows. See also MDI child , page 331 and MDI parent , page 331.

Term	Definition (continued)
Natural Construct nucleus	Sophisticated driver program that invokes the model subprograms at the appropriate time in the generation process and performs the functions common to all models (for example, using windows and performing PF-key functions). The nucleus communicates with the model subprograms through standard parameter data areas (PDAs). These PDAs contain fields assigned by Natural Construct, as well as fields that are redefined as required by a model.
Natural Debugging facility	The Natural Debugging facility is one of the utilities available in a Natural environment. Construct Spectrum users gain access to the facility using the Invoke Proxy function in the Construct Spectrum Administration subsystem. The subprogram proxy then sets up an online environment which simulates the client/server environment. The simulation allows you to use all the features of the Natural Debugging facility.
navigation bar	A menu bar on a web page used to navigate to other pages or perform actions.
node	An individual computer or, occasionally, another type of machine in a network.
nucleus	See Natural Construct nucleus , page 333.
object	Any application component including a form or a record. A business object is a group of services related to a common business entity such as Customer, Order, and Department.
object factory	Visual Basic module that identifies all objects and methods within an application and instantiates objects upon request.
object factory wizard	A Visual Basic Add-In that updates an object factory in a Construct Spectrum web application.

Term	Definition (continued)
object library	Provides definitions for all the objects, methods, and properties an OLE automation server exposes. Equivalent to type library , page 338.
OLE automation server	OLE is the acronym for object linking and embedding. The OLE automation server is a code component that passes objects to other applications so that these applications can programmatically manipulate the objects.
overflow condition	Situation where there are more fields than can be displayed on a maintenance dialog.
package	Collects all of the modules necessary to implement a business object. A package combines components and classes to provide both browse and maintenance services for a database table. A package is composed of a set of modules generated from a multi-module generate. An application is made up of one or more packages.
page handler	A class that exchanges replacement tags in an HTML template with other content.
page handler wizard	Construct Spectrum Add-In to Visual Basic that generates page handlers for web applications.
parent model	A model that collects parameters for child models and generates specifications; a Super model.
parse area	Code in the page handler that deals with replacement tags.
ping	A request that is sent to a service to determine whether the service is running.
platform	A piece of equipment that, together with its operating system, serves as a base on which developers can build other systems. For example, an MVS mainframe computer can serve as a platform for a large accounting system.
project	The collection of files used to build an application in Visual Basic. See also group project.

Term	Definition (continued)
project window	Lists the forms and modules in the current Construct Spectrum project in Visual Basic.
properties window	Lists the property settings for the selected form or control in Visual Basic.
property	A characteristic of an object, such as size, caption, or color. In Construct Spectrum, this refers to an object's data settings or attributes in Visual Basic.
regenerate/preserve status	Each code block in a module is marked to be either regenerated or preserved. Blocks marked to be regenerated will be replaced or deleted during a regenerate. Blocks marked to be preserved will be left alone.
remote call	Communication with an object residing in a different location, such as a server.
replacement tag	HTML tag that is replaced with content or HTML containing additional replacement tags when the web page is assembled. Some replacement tags are used to remove existing sections of HTML. For example, a security tag can be used to specify content that only certain users can access.
resource	Text or binary value that can be localized. See also localization , page 330.
run	Execute or invoke a module or application.
security cache	A file used to store recently-accessed security data.
server	A physical computer that provides services to another computer (called a client) and responds to a request for services. On multitasking machines, a process that provides services to another process is called a server.
server application	An application that runs on a server machine.

Term	Definition (continued)
service	A software service that runs on a server. There could be several services running on each server.
service exit	An exposed exit routine that is called by the Spectrum dispatch service and can be replaced by a user-supplied routine.
service log	A file used to store service log data.
shutdown	A command that is sent to a service to terminate that service.
Construct Spectrum client/server application	An application created with Construct Spectrum wizards and Add-Ins. Users access mainframe business functions and data through a Visual Basic component running on a Windows platform.
Spectrum Control record	A record that is created daily that contains system control and statistic data for a Spectrum dispatch service.
Spectrum Dispatch Client (SDC)	Provides the Construct Spectrum data exchange that facilitates calls from a client to Natural subprograms running on a server.
Spectrum dispatch service	Middleware component used to encapsulate broker calls on the server, provide directory services, enforce security, and invoke backend Natural services.
Spectrum security service	A component of the Construct Spectrum Administration subsystem that controls group (and therefore user) access to application libraries, objects, and methods.
Spectrum Service Manager	An client tool supplied with Construct Spectrum that allows users to specify the Spectrum services that the client will use to communicate with the server.
Spectrum service settings	A collection of parameters used to configure a Spectrum service.

Term	Definition (continued)
Construct Spectrum web application	An application created with Construct Spectrum wizards and add-ins. Users access mainframe business functions and data through a web browser.
Construct Spectrum web framework	A group of Visual Basic modules and classes that collaborate to dynamically generate web pages.
Status bar	In clien/server application, displays status information about a selected item, application, or business object, such as the status bar included on maintenance dialogs. The status bar contains sections for a message, status indicators, and the current date and time. Status bars also appear at the bottom of an MDI form.
steplib chain	A hierarchy of Natural libraries that determines the location from which modules are executed.
Sub Main procedure	The first Visual Basic procedure to be executed when a Construct Spectrum application is run. Each Visual Basic application has one Sub Main procedure.
subprogram proxy	Natural subprogram called by a Spectrum dispatch service, which in turn calls a target Natural subprogram. Each subprogram requires a subprogram proxy to translate between network and Natural data formats, allowing Construct Spectrum to provide a common interface to all subprograms.
Super model	A model capable of generating multiple components necessary for a Construct Spectrum client/server or web application using a minimum of input parameters. To generate multiple components, a Super model creates model specifications using all of the models required to create the individual components. The modules generated by a Super model make up a package. See also package , page 334.
target module	See target subprogram , page 338.
target object	See target subprogram , page 338.

Term	Definition (continued)
target subprogram	Any Natural subprogram.
template parser	A class used to parse HTML or other templates.
toolbar	Contains three toolbar buttons that provide quick access to commonly used commands in the application. The user clicks the appropriate button on the toolbar to perform the action it represents. Any action that can be performed from a toolbar button can also be invoked from a menu.
toolbar button	An icon on the toolbar that allows users to perform an action.
toolkit	A set of related and reusable classes designed to provide useful, general-purpose functionality. An application will usually incorporate classes from one or more libraries of predefined classes called toolkits or software development kits (SDKs). An example of a toolkit is a set of collection classes for lists, associative tables, and stacks. Toolkits emphasize code reuse. They are the object-oriented equivalent of subroutine libraries.
Trace options	Options that specify how to trace messages sent between the client and server.
type library	Provides the definitions for all objects, methods, and properties exposed by the OLE automation server. This is the equivalent to object library , page 334.
upload	Process of transferring modules from the client to the server.
variant	A Visual Basic term identifying a late-binding data type. Variants allows Construct Spectrum subroutines or functions to accept different types of data and determine the exact type when they receive the value in Visual Basic.

Term	Definition (continued)
VB-Client-Server-Super-Model	A model that can generate all of the modules required for a fully functional client/server application. The Super model can generate all of the modules required for maintenance and browse services for up to 12 business objects at a time. See also Super model , page 337.
verification rules	Business rules defined in Predict that are implemented in the object subprogram on the server and in the VB-Maint-Object on the client. Verification rules also help provide default values for fields that are derived to be represented by GUI controls such as CheckBoxes, Option buttons, or drop-down combo boxes. For example, verification rules will force users to make a selection based on one or more choices. If the application has a field where a valid state name must be entered, a verification rule can be attached to the field so that only valid state names will be accepted.
Visual Basic browse object	A Visual Basic class designed to configure an instance of a BrowseBase class. This class delivers information about the columns and keys supported by the browse subprogram to the client framework which, in turn, configures and displays the browse dialog at runtime. See also Visual Basic business object , page 339.
Visual Basic business object	Conceptual object with a domain on the client that is used to implement client business rules and encapsulate communication with the Construct Spectrum Client. Comprised of class modules or objects. Includes Visual Basic browse objects and Visual Basic maintenance objects.
Visual Basic maintenance object	A Visual Basic class instantiated by a maintenance dialog to encapsulate calls to the Construct Spectrum Dispatch Client and to implement validation in the maintenance dialog. See also Visual Basic business object , page 339.
web super wizard	A Construct Spectrum Add-In to Visual Basic that generates multiple HTML templates and page handlers for a web application.

Term	Definition (continued)
web class	A visual basic class that responds to urgent requests (ASP requests) for a web page.
WebClass ASP	An ASP script used to instantiate a web class.
wildcard	A character or symbol (for example, “*”, “<”, or “>”) that qualifies a selection. The asterisk (*) wildcard character indicates the unique portion of the file names selected. For example, when using the Construct Spectrum Download dialog, you could use a wildcard character to list all modules that begin with “Maint” by typing “Maint*” in the selection box.
XML extract	An extract of information stored in Predict and other information stored on the client as XML meta data. Includes information such as business objects and formatting that is used by wizards to build application components.

APPENDIX B: UTILITIES

This chapter explains the utilities supplied with the Construct Spectrum Administration subsystem.

The following topics are covered:

- **SPUREPLY Subprogram**, page 342
- **SPUETB – Spectrum Interface Subprogram**, page 350
- **The Conversation Factory**, page 363
- **SPUTLATE — Character Translation**, page 364
- **Multi-Tasking Verification Utility**, page 365
- **Log Utilities**, page 366

SPUREPLY Subprogram

SPUREPLY is mainly used by servers to send responses back to a client. The response can be defined as a SYSERR message or it can be hardcoded.

Features and Benefits

SPUREPLY has the following benefits and features:

- Defines a standard protocol for exchanging messages
- Enables messages to be multi-lingual if you define them in SYSERR
- Automatically performs message substitution of :1:, :2:, and :3: within the SYSERR messages
- Can send other information in addition to a message

Limitation

The maximum supported response length is 5000 bytes.

Supported Methods

SPUREPLY supports five methods, as defined in SPLREPLY. One of these methods must be assigned to the SPAREPLY.METHOD parameter before calling SPUREPLY.

Method	Description
SEND-REPLY	Normal method used to send a single message reply. This causes the message to be sent with the End of Conversation option.
SEND-WITHOUT-EOC	This method is used to send a multi-part reply. The SEND-REPLY method should be used when sending the last message of the reply.
LOOKUP-MESSAGE	This method is used when you only want to look up the error message text but not send it.

Method	Description (continued)
SEND-MESSAGE-ONLY	Used to send the message text without the standard protocol information.
SEND-MESSAGE-ONLY-WITHOUT-EOC	Same as SEND-MESSAGE-ONLY but does not include End of Conversation option.

Message Protocol

All messages sent to the client use the following protocol:

Message	Protocol
SIGNATURE(A6)	Constant MSG111. This defines the structure of the send buffer.
RESPONSE-CODE(N4)	Response code passed to SPUREPLY in SPAREPLY. Successful responses should use a response code of zero. Other predefined Response codes are: 001 - Replica ID was not matched 9999 - Natural runtime error
REPLICA-ID(A32)	Replica ID passed in from SPAETB.
SPECTRUM-SERVICE(A32)	Passed from SPAETB
SYSERR-LIBRARY(A8)	The name of the SYSERR library where the message was looked up.
MSG-NR(N4)	The number of the SYSERR message.
MESSAGE(A1/1:V)	Message portion of send buffer. In most cases, this message area contains a message looked up in SYSERR by SPUREPLY. Additional information can also be passed in this area.

Call Interface

SPUREPLY supports the following interface:

```
PARAMETER USING SPAREPLY      /* Specific parameters
PARAMETER                     /* The message portion of the send buffer
01 SPAREPM
    02 INPUT-OUTPUTS
        03 BUFFER-LENGTH (I2)
        03 MSG-BUFFER (A1/1:V)
PARAMETER USING SPAETB        /* Parameters to SPUETB
PARAMETER USING ETBCB         /* Standard broker control block
PARAMETER USING CDPDA-M       /* Standard message area
```

These data areas are described in the following section.

Data Areas

SPAREPLY Data Area

SPAREPLY is passed to SPUREPLY. It contains the following data:

Parameter	SPAREPLY	Library	S421	DBID	17	FNR	60
Command							> +
I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment	
Top	-			-			
*							
*							
*			Data Area Name: SPAREPLY			Function	
*			Created on....: Jun 12, 98			=====	
*			Created by....: SAG			This data area is passed to	
*						SPUREPLY which is used to	
*						send a reply back to a client.	
*							
*						The reply structure is	
*						defined in SPLREP.	
	1		SPAREPLY				
	2		INPUTS				
	3		METHOD	I	1	/* See SPLREPLY	
	3		RESPONSE-CODE	N	4	/* This response, use zero for	
*						/* successful response.	
	3		SYSERR-INFO				
	4		MSG-NR	N	4	/* SYSERR Message number	
	4		SYSERR-LIBRARY	A	8	/* Defaults to SYSSPEC	
	4		MSG-DATA	A	32	(1:3) /* Subs. values	
*						/* May contain *NNNN references	
	3		TRANSLATE	L		/* Translate character set. If	
*						/* currently EBCDIC, message	
*						/* will be translated to ASCII	
*						/* and vise-versa.	
	3		EMBEDDED-MSG- INFO			/* This structure is only used	
*						/* when the message to be looked	
*						/* up is only a portion of the	
*						/* data to be sent. In this case	
*						/* you must indicate where the	
*						/* message is in the send buffer	
	4		MSG-START	I	2	/* Byte location of start of msg	
	4		MSG-LENGTH	I	2	/* Total length of message	
						/* portion	

The fields of SPAREPLY are described in the following table:

Field name	Description
METHOD (I1)	Assign a value from SPLREPLY to indicate whether you want to perform a send with EOC, send without EOC, or just look up the message text.
RESPONSE-CODE (N4)	Assign the response code value to be sent to the client.
MSG-NR (N4)	If a message is to be looked up in SYSERR, assign the message number to this field.
SYSERR-LIBRARY (A8)	By default, all messages are looked up in the library SYSSPEC. If this is not the desired library, specify an alternative here.
MSG-DATA (A32/1:3)	Up to three values can be specified for substitution into the message. These values will replace the :1:, :2:, and :3: placeholders in the SYSERR message. Optionally, the substitution values themselves can be looked up in SYSERR by specifying message data in the format *nnnn.
TRANSLATE (L)	If you want the message to be translated (from EBCDIC to ASCII or vice versa), mark this flag.
MSG-START (I2)	If the message retrieved from SYSERR represents only a portion of the data to be sent, indicate the starting position of the message portion of the send buffer here.
MSG-LENGTH (I2)	This field is only required when MSG-START is assigned. Indicate the length of the message portion of the send buffer.

SPAREPM Data Area

SPAREPM is an example of a standard message area that can be passed to SPUREPLY. This data area can be used to send messages up to 250 characters in length. After SYSERR messages are looked up, the resulting message text is returned in this parameter. The values in SPAREPLY.MSG-START and SPAREPLY.MSG-LENGTH determine where the message will be assigned. If these values are zero, the message will be returned starting at position 1 for a length of SPAREPM.BUFFER-LENGTH.

The data area contains these fields:

```

cal      SPAREPM   Library S421                      DBID    17 FNR    60
Command
I T L Name                                     F Leng Index/Init/EM/Name/Comment
All - -----
*   Data Area Name: SPAREPM                      Function
*   Created on....: Jun 12, 98                      =====
*   Created by....: SAG
*
*   This data area can be used as
*   the second parameter to
*   SPUREPLY. When a message number
*   is passed to SPUREPLY, the
*   message text is returned in
*   this parameter.
*   Alternatively, the message to
*   be sent can be passed to
*   SPUREPLY using this parameter.
1 SPAREPM
2 INPUT-OUTPUTS
3 BUFFER-LENGTH          I    2 INIT<250>
3 MSG-BUFFER             A    1 (1:250)
R 3 MSG-BUFFER
4 MSG-STRING             A    250
----- Current Source Size: 1201 Free: 100104 ----- S 17 L 1

```

If you want to send information other than a standard message, copy SPAREPM and define the fields you need to send (up to 5000 bytes). Be sure to assign the BUFFER-LENGTH field to reflect the size the data to be sent.

Call Example

```

/*
/* SYSSPEC/1001: Invalid request:1:sent to:2:expecting:3:
ASSIGN SPAREPLY.MSG-NR = 1001
ASSIGN SPAREPLY.MSG-DATA(1) = #COMMAND
ASSIGN SPAREPLY.MSG-DATA(2) = *PROGRAM
ASSIGN SPAREPLY.MSG-DATA(3) = '''CREATE'''
ASSIGN SPAREPLY.RESPONSE-CODE = 1 /* Invalid command
PERFORM SEND-MESSAGE
*
*****
DEFINE SUBROUTINE SEND-MESSAGE
*****
*
  IF #I-AM-ASCII NE #CLIENT-IS-ASCII THEN
    ASSIGN SPAREPLY.TRANSLATE = TRUE
  END-IF
  ASSIGN SPAREPLY.METHOD = SPLREPLY.SEND-REPLY /* Send with eoc
  CALLNAT 'SPUREPLY' SPAREPLY
              SPAREPM
              SPAETB
              ETBCB
              MSG-INFO
END-SUBROUTINE /* SEND-MESSAGE

```

Send Buffer Example

```

MSG1110001ATTACH-MANAGER--B0B218EC55E1AE01          AURORA-CONVERSATION FACTORY
SYSSPEC 1001Invalid request CMD SH sent to SPSCFACT expecting 'CREATE'

```

Where...

MSG111	is the message signature
0001	is the response code
ATTACH-MANAGER-- B0B218EC55E1AE01	is the server replica ID
AURORA-CONVERSATION-FACTORY	is the spectrum service

Where...	(continued)
SYSSPEC	is the name of the SYSERR library used
1001	is the SYSERR message number
Invalid request CMD SH sent to SPSCFACT expecting 'CREATE'	is the message text

SPUETB – Spectrum Interface Subprogram

Writing robust servers can be a very complex task. There are many possible errors that can occur, and ensuring that each error is handled in the proper way is very difficult. Some errors are caused by resource shortages, so it is desirable to retry the call again after a brief pause. Other errors are fatal and should result in the server shutting down. Still other errors, like wait timeouts, are normal and expected.

To help simplify and standardize the task of writing servers, Construct Spectrum includes a subprogram that wraps the Broker ACI calls. This wrapper subprogram, called SPUETB, handles many situations that have to be coded for when making direct Broker calls. We recommend that all broker calls be made using SPUETB to ensure that errors are handled and logged properly.

Features and Benefits

Here is a summary of the capabilities offered by SPUETB:

Broker Error Handling

Most Broker errors are handled internally by SPUETB. If the errors are due to resource shortages, SPUETB will pause for two seconds and then try the call again. SPUETB will continue to retry the call for up to twenty seconds.

When implementing server receive loops, SPUETB handles all wait timeouts (Broker error 74) and returns to the receive state.

Fatal errors will cause the server to shutdown if SPUETB is granted shutdown permission.

SPUETB can also handle message length errors and return a message to the sender of the message indicating that the sent message was too long.

Error Logging

All errors that are returned from Entire Broker are logged on the Spectrum Communication Log. This log can be used to help detect problems with your programs or environment.

Shutdown Requests

SPUETB will automatically respond to shutdown requests from Entire Broker. These requests can be initiated using the EntireX Broker Control Center.

Server Timeouts

Whenever the server has not received a message for the length of time specified on the service record, the server automatically shuts down.

Command Handling

SPUETB automatically registers for the CMD service and responds to all command requests. Command requests include the CMD CALLNAT command which allows you to supply the name of the subprogram to be called.

SPUETB Interface

SPUETB is called using the following interface:

```
DEFINE DATA
  PARAMETER USING SPAETB      /* Specific Parameters
  PARAMETER USING ETBCB       /* Standard broker control block
  PARAMETER
    01 SEND-BUFFER(A1/1:V)
    01 RECEIVE-BUFFER(A1/1:V)
    01 RESERVED-AREA(A1/1:V) /* Reserved for SPUETB use
  PARAMETER USING CDPDA-M     /* Standard message area
END-DEFINE
```

For most calls, the caller is responsible for filling in the Broker Control Block as would be done for a direct call to Entire Broker. Additionally, the caller can specify the degree of error handling and support for common functions that should be handled by SPUETB.

The data areas are described in the following sections.

Data Areas

SPAETB Data Area

```

Parameter SPAETB      Library S421
Command
I T L Name                      F Leng Index/Init/EM/Name/Comment      > +
Top - -----
*
*
*   Data Area Name: SPAETB                      Function
*   Created on....: May 05, 98                  =====
*   Created by....: SAG                        This data area is passed to
*                                              SPUETB which is used to
*                                              encapsulate calls to Entire
*                                              Broker. Use SPLETB to assign
*                                              constant values.
*
1 SPAETB
2 FORCE-PDA                      A    1 (1:V) /* This field is only here
*                                     /* to force the caller to create
*                                     /* a separate LDA to call SPUETB
*                                     /* rather than using SPAETB.
*                                     /* This way, initial values can
*                                     /* be placed in the LDA so that
*                                     /* defaults get assigned.
2 INPUTS
3 METHOD                          I    1 /* 0 = Normal call
*                                     /* See SPLETB for other methods
*                                     /* Set desired functions ...
3 ENCAPSULATED-FUNCTIONS
4 SUPPORT-SERVER-COMMANDS       L
*                                     /* SPUETB will automatically
*                                     /* register a command service
*                                     /* whenever a regular service is
*                                     /* registered. CMD is used as
*                                     /* the broker service name.
*                                     /* SPUETB will handle all
*                                     /* command requests directly.
4 ALTER-RECEIVE-SERVICE        L
*                                     /* Automatically change the
*                                     /* service name on receive to
*                                     /* an '*' to allow commands
4 SHUTDOWN-PERMISSION          L
*                                     /* If true, SPUETB is allowed to
*                                     /* shutdown the server directly.
*                                     /* See SHUT-DOWN-REASONS
4 SHUTDOWN-REASONS              /* Set desired shutdown reasons:
*                                     /* only set after method 6
*

```



```

5 EXPLICIT-SHUTDOWN          L      /* Shutdown request from BROKER
*                               /* or from Spectrum console.
5 TIMEOUT-REACHED           L      /* See TIMEOUT-HANDLING
5 TERMINAL-ERROR             L      /* Non-recoverable broker error.
4 TIMEOUT-HANDLING           I      4 /* 0 = Return all timeouts so
*                               /* that caller can handle
*                               /* >0= Reissue call for this
*                               /* many seconds. Set to
*                               /* max desired idle period.
*                               /* -1= Reissue call indefinitely
*                               /* -1 is normally used by
*                               /* ATTACH servers which
*                               /* should run forever.
3 ERROR-HANDLING
4 HANDLE-TRUNCATION-ERROR    L      /* SPUETB will respond to
*                               /* ETB error 00200094. This
*                               /* won't be sent back to caller
4 RESERVED                   A      8 /* Reserved for future.
4 USE-SPECTRUM-ERROR-LOG     L      /* Log all errors on the Spec.
*                               /* file. Warning, this will
*                               /* cause an ET to be issued.
4 WRITE-ERRORS-TO-CONSOLE    L      /* CALL 'CMWTO' with errors
4 WRITE-ERRORS-TO-PRINT-FILE-0 L      /* Write errors to Natural
*                               /* print file 0
4 MAX-RETRY-TIME             I      2 /* Number of seconds to continue
*                               /* to retry call in the event of
*                               /* a Broker resource shortage.
*                               /* Defaults to 20 seconds.
4 MESSAGE-DATA               /* These fields are used to
*                               /* build helpful error messages
*                               /* when broker calls fail.
5 CALLING-PROGRAM            A      8 /* Name of caller.
5 SPECTRUM-SERVICE           A      32 /* Name of spectrum service
*                               /* if known.
5 CALL-DESC                  A      32 /* Description of the call
2 INPUT-OUTPUTS
3 REPLICA-ID                 A      32 /* Assigned at first LOGON
*                               /* do not adjust
3 CLIENT-MODE                L      /* In this mode, errors need
*                               /* not be logged and checks
*                               /* for broker error cycles
*                               /* are not performed.
3 OPTION                     A      50 /* SPUETB option
2 OUTPUTS
3 RESULT                     I      1 /* See SPLETB
*                               /* 0 = Normal request
*                               /* 1 = Attach request
*                               /* 2 = Command request
*                               /* 3 = Timeout
*                               /* 4 = Non-terminal error
*                               /* 5 = Terminal error
*                               /* 6 = Restarting after error.

```

The following table provides additional details about these fields:

Field name	Description
FORCE-PDA (A1/1:V)	<p>Due to the number of input settings that must be assigned before calling SPUETB, the preferred method of assigning them is to use supplied LDAs, initialized with common defaults settings. The following LDAs are supplied:</p> <ul style="list-style-type: none"> • SPAETBC is used by Broker Client programs • SPAETBS is used by Broker Server programs
METHOD (I1)	This field determines the type of processing to be performed by SPUETB. The method values should be assigned using one of the constants in SPLETB.METHODS.
SPLETB.NORMAL-CALL	Used for all broker calls except LOGON, REGISTER, DEREGISTER, and LOGOFF.
SPLETB.LOGON	This method uses the value of SPAETB.SPECTRUM-SERVICE to look up the Broker ID, User ID, and corresponding password with which to log on to Entire Broker. It also executes the Broker Logon function.
SPLETB.REGISTER-SERVER	Uses SPAETB.SPECTRUM-SERVICE service to look up the Broker ID, Server Class, Server Name, and Service and uses these to Register with Entire Broker. If the SUPPORT-SERVER-COMMANDS parameter is set to TRUE, this method also registers an additional service, CMD, upon which to accept commands.
SPLETB.SHUTDOWN-SERVER	This method is for use by Servers only. It invokes the Broker Deregister and Logoff functions. A shutdown request should always be issued before ending your server programs. You can tell SPUETB to perform a shutdown automatically by assigning the parameter SHUTDOWN-PERMISSION to TRUE.

Field name	Description (continued)
SPLETB.LOG-SUPPLIED-ERROR	This method is used to request that an application error be logged by SPUETB. The error to be logged must be passed in the MSG-INFO.##MSG field of data area CDPDA-M. The message will be logged to locations specified under the SPAETB.ERROR-HANDLING structure.
SPLETB.LOG-Natural-ERROR	This method should only be called from ON ERROR blocks or error transactions (assign to *ERROR-TA). This tells SPUETB to log the last Natural error that occurred. The error will be logged to locations specified under the SPAETB.ERROR-HANDLING structure.
SPLETB.GET-SERVICE-DEFAULTS	This method is used to assign the following fields based on the values established at the time of the initial LOGON method. ETBCB.BROKER-ID, ETBCB.SERVER-CLASS, ETBCB.SERVER-NAME ETBCB.SERVICE, ETBCB.USER-ID, ETBCB.TOKEN, ETBCB.SECURITY-TOKEN SPAETB.TIMEOUT-HANDLING, SPAETB.SPECTRUM-SERVICE.
SPLETB.LOGOFF	Performs a Broker Logoff function. For other methods, refer to SPLETB.
SUPPORT-SERVER-COMMANDS (L)	This tells SPUETB to automatically support command services such as PING, SHUTDOWN, etc. SPUETB will automatically register a separate service using CMD as the service name. All command requests are handled automatically by SPUETB, the caller need not code any specific support for commands.
ALTER-RECEIVE-SERVICE (L)	This field is used in conjunction with SUPPORT-SERVER-COMMANDS. If this field is set to true, SPUETB automatically changes the service name specified on any receive function to an asterisk (*). This allows the receive to be satisfied by either a request for the main service or a request for the command service.

Field name	Description (continued)
SHUTDOWN-PERMISSION (L)	If true, SPUETB is allowed to shutdown the current program. This is normally only set for server programs. Assign the fields within SPAETB.SHUTDOWN to determine which events allow SPUETB to shut down the running server. SPUETB always logs any errors prior to shutting down.
EXPLICIT-SHUTDOWN (L)	Allow SPUETB to shut down the server as a result of an explicit SHUTDOWN command.
TIMEOUT-REACHED (L)	Allow SPUETB to shut down the server when the server timeout value has been reached. This timeout value is passed in parameter TIMEOUT-HANDLING and is defaulted from the Server Timeout field on the Spectrum Service Record.
TERMINAL-ERROR (L)	Allow SPUETB to shut down the server in response to a fatal Broker error.
TIMEOUT-HANDLING (I4)	<p>This field tells SPUETB how to handle timeouts when executing Broker RECEIVE functions. The following values are supported:</p> <ul style="list-style-type: none"> • -1 Execute forever (use SPLETB.NO-TIME-LIMIT) to assign this value. • 0 Return to the caller after the first receive timeout • >0 Execute for this many seconds, then either return to the caller or execute shutdown processing (based on SHUTDOWN-PERMISSION and TIMEOUT-REACHED parameters).

Field name	Description (continued)
	<p>The TIMEOUT-HANDLING field is derived from the Server Timeout value on the Spectrum Service record. If no server timeout is specified, the following defaults are used:</p> <ul style="list-style-type: none"> • Services without Attach Servers -1 • Services with Attach Servers 1200 (= 20 minutes)
RESERVED (A8)	This field is reserved for future use.
USE-SPECTRUM-ERROR-LOG (L)	Log all errors to the Spectrum log file.
WRITE-ERRORS-TO-CONSOLE (L)	Write all errors to the operator console.
WRITE-ERRORS-TO-PRINT-FILE 0(L)	Write all errors to Print file 0.
MAX-RETRY-TIME (I2)	This is the length of time to continue trying to execute a Broker call in the event of a Broker resource shortage. This defaults to 20 seconds.
CALLING-PROGRAM (A8)	This identifies the caller of SPUETB. This name is used when logging error messages.
SPECTRUM-SERVICE (A32)	In order to use the Logon and Register methods of SPUETB, you must supply the name of the Spectrum service in this field. This name is also used when writing out error messages.
CALL-DESC (A32)	This is a free-format description of the call to be used when logging error messages.
REPLICA-ID(A32)	The replica id assigned to the server. This is an output field.

Field name	Description (continued)
CLIENT-MODE(L)	If this flag is set, SPUETB will not log errors, and checks for broker error cycles are not performed.
RESULT (I1)	<p>The result field should be interpreted after the call to determine the results of the call. The data area SPLETB defines the following constants which should be used to check the results:</p> <ul style="list-style-type: none"> • NORMAL-REQUEST Broker call completed normally. • ATTACH-REQUEST Broker call resulted in an Attach request. This would only be returned to Attach Services. • TIMEOUT Receive timeout was reached, and shutdown permission for timeouts was not granted to SPUETB. • NON-TERMINAL-ERROR Non-terminal broker error has occurred. This error is automatically logged by SPUETB. See ETBCB.ERROR-CODE. • TERMINAL-ERROR Terminal Broker error occurred, but shutdown permission was not granted to SPUETB. The error is automatically logged.

ETBCB Data Area

ETBCB is a standard data area representing the fields that must be passed to Entire Broker when using the Broker ACI. The calling program should use either ETBCB12 or ETBCB13, depending on the version of the Broker stub they are using.

SEND-BUFFER

The send buffer is used in conjunction with the Broker Send function. The size of this buffer must be greater than or equal to the value of ETBCB.SEND-LEN.

RECEIVE-BUFFER

The receive buffer is used in conjunction with the Broker Receive function or blocked Sends. The size of this buffer must be greater than or equal to the value of ETBCB.RECEIVE-LEN.

RESERVED-AREA

This pass area is reserved for future use. Define and pass the parameter SPAET-BP.NOT-USED(*) in place of this parameter.

CDPDA-M

This is a standard message area. Whenever SPUETB encounters a non-recoverable error, it returns with the error text in MSG-INFO.##MSG and MSG-INFO.##RETURN-CODE is assigned “E”.

Using SPUETB

For an example of using SPUETB, refer to the Timestamp Server example SP-STIMS. If you need to do your own character set translation (because your messages contain a mixture of printable and binary data), refer to SPSTIMS2.

CMD TRACE

The TRACE command is used to enable and disable tracing of a running server. This feature is used in conjunction with the CSUDEBI utility. The TRACE command accepts a RID to target the command to a specific replica.

There are two separate forms of the TRACE command; the one you choose depends on whether you wish to enable or disable tracing.

Enabling Tracing

To enable tracing use the command `CMD TRACE LOCATION=n [options]`

The valid trace locations are defined in the LDA CSLDEBUG (in SYSCST). The following table shows the valid options:

Keyword	Description
QHANDLE	A valid queue handle is required when setting the message location to 10. This is a quoted value consisting of 'bkrid, user-ID, token, (unpacked) security-token, conv-ID'
ERROR-TRIGGER	<p>This is used to force a runtime error at a specified point within the running server. Errors can only be triggered on lines that are currently being traced. The syntax of the value assigned to this field is:</p> <p><code>Program,Line,NATnnnn,Skip'</code></p> <p>where <code>Program</code> is the name of the program where the runtime error is to be triggered, <code>Line</code> is the line number where the error is to be triggered, and <code>NATnnnn</code> is the error to be triggered. <code>Skip</code> is used if the error is not to be triggered on the next execution of the statement, but rather after executing the statement this many times.</p>
FILTER-MASK	A100 string of 0 and 1 values. "1" is used to represent statements that are to be traced. Each mask character is related to a constant in the LDA SPLTRACE.
FILTER-PROGRAM	A list of up to five programs (in quotes and separated by commas) used to limit the programs that produce trace output. Special characters can be used in the program name to serve as pattern-matching characters. For details of these special characters, see the PATTERN option of the Natural EXAMINE statement.

Example

```

CMD TRACE
RID=BBCB0B5A1BD5AF9F201FACB0B5A14D5AF9F201, LOCATION=10, QHANDLE='BKR045
, SPSCFACT, AAC
B0B5A1BD5AF9F201FACB0B5A14D5AF9F201, 00000000000000000000000000000000
00000000000000000000000000000000, 00000000000000000000000000000000, ERROR-
TRIGGER='SPUETB, 5420, NAT0082', FILTER-MASK=10001100000000010000
0001000000001100000000000000000000000000000000000000000000000000, FILTER-
PROGRAM='SPU*, SP?SEC'

```

Disabling Tracing

To disable tracing, use the command `CMD TRACE OFF`.

Trace Response

The trace response is normally a confirmation message indicating whether or not the trace request was successful. The response uses the `SPUREPLY` protocol (MSG111).

Testing Trace Facility

The trace functions can be tested by using the command `CMD CALLNAT SPUTRTST`.

CMD CALLNAT

It is possible to `CALLNAT` any subprogram, provided it implements a generic interface. This interface is defined as follows:

```

DEFINE DATA
  PARAMETER USING SPACALLN /* Standard callnat parameters
  PARAMETER USING SPAREPLY /* Reply message parameters
  PARAMETER
    01 RECEIVE-SEND-BUFFER(A1/1:15000)
END-DEFINE

```

The `CALLNAT` command takes the form:

```
CMD CALLNAT subpname parameter_string
```

Where...

Subpname	is the name of the subprogram to be CALLNATed.
parameter_string	is any set of characters to be passed to the specified subprogram using the RECEIVE-SEND-BUFFER.

See the subprogram SPUCMDT as an example of how to write a new CALLNAT interface subprogram.

The Conversation Factory

Construct Spectrum includes a facility known as a Conversation Factory. The Conversation Factory works in conjunction with high-level callnat and message queue APIs to facilitate the simple transfer of data between two platforms. Here is a summary of the benefits offered by the Conversation Factory and supporting APIs:

- Allows communication between a client and a server without knowledge of the Broker ACI.
- Allows a conversation to be established between two processes, each acting as clients.
- Supports multiple concurrent conversations between the same two participants. For example, the Construct generate server listens for Specifications on one conversation and cancels requests on another.
- Used in conjunction with servers that are launched from the client in order to establish a conversation between the client who launched a service and the service itself.

On the server, the Conversation Factory consists of the following four subprograms:

Subprogram	Description
SQUOPEN	Used to open a new conversation.
SQUSEND	Used to send information from one end of the conversation to the other.
SQUIRECV	Used to receive information.
SQUCLOSE	Used to close the conversation.

For an example of how to use the Conversation Factory APIs, refer to the subprogram SQEXAMPL.

SPUTLATE — Character Translation

When writing your own servers, it is sometimes necessary to perform character-set translation. The preferred approach to character translation is to use the translation routines that can be assigned to the Broker Service in the Broker Attribute File. However, sometimes you may want to send a message that contains a mixture of binary and printable data where only a portion of the message is to be translated. Use the subprogram SPUTLATE for this purpose.

SPUTLATE allows you to pass in a string, along with an array of character positions to be translated. SPUTLATE is supplied in source form. For an example of calling SPUTLATE, see SPSTIMS2.

Determining a Character Set — SPUASCII

Sometimes a server receives a message it cannot interpret. Normally, the server returns a reply to the sender indicating that the message is invalid. If the server performs its own translation, it needs to know the character set of the received message so that the reply can be sent back in the client's character set. SPUASCII can be used to help determine whether a string of characters appears to be in ASCII or EBCDIC format. For an example of calling SPUASCII, see SPSTIMS2.

Multi-Tasking Verification Utility

Use this utility to verify that your batch Natural nucleus is a re-entrant nucleus and to verify whether ADALNK has been configured to be re-entrant. Both of these conditions are required to run Spectrum services in a batch multi-tasking environment.

Module TESTTASK can be used to start multiple Natural subtasks. Module TESTSTSK can be used as the subtask that is started by TESTTASK. If your batch nucleus or ADALNK is not re-entrant, the job that runs TESTTASK will not end. Otherwise, tracing is written in the job output showing execution status of the subtasks.

Log Utilities

Construct Spectrum supplies several utilities for archiving and deleting log data.

Spectrum Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description
BSBLARCP	Allows the Spectrum Log data to be archived to a workfile and optionally deleted from the Spectrum Log based on a date. It also generates a log record of the archive process. This utility has the following input fields: Input End Date Enter the last LOG date to be archived. This parameter applies to all log archive utilities. Full Report Enter "F" to display the full details of all the data being logged. Enter "B" to show only the main log information. This parameter applies to all log archive utilities. Delete After Archive Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
BSBLRESP	Restores data to the Spectrum Log file. It uses the entire log data created by the BSBLARCP utility. It also generates a log record of the restore process. This utility has the following input field: Full Report Enter "F" to display the full details of all the data being logged. Enter "B" to show only the main log information. This parameter applies to all log archive utilities.

Construct Spectrum Control Record Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description						
BSCTARCP	<p>Allows the Spectrum Control Record log data to be archived to a workfile and optionally deleted from the Spectrum Control Record log, based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p> <table><tr><td>Input End Date</td><td>Enter the last LOG date to be archived. This parameter applies to all log archive utilities.</td></tr><tr><td>Full Report</td><td>Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.</td></tr><tr><td>Delete After Archive</td><td>Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.</td></tr></table>	Input End Date	Enter the last LOG date to be archived. This parameter applies to all log archive utilities.	Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.	Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
Input End Date	Enter the last LOG date to be archived. This parameter applies to all log archive utilities.						
Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.						
Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.						
BSCTRESP	<p>Restores data to the Spectrum Control Record log file. It uses the entire log data created by the BSCTARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p> <table><tr><td>Full Report</td><td>Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.</td></tr></table>	Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.				
Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.						

Domain Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description
BSDOARCP	Allows the Domain log data to be archived to a workfile and optionally deleted from the Domain log based on a date. It also generates a log record of the archive process. This utility has the following input fields: Input End Date Enter the last LOG date to be archived. This parameter applies to all log archive utilities. Full Report Enter "F" to display the full details of all the data being logged. Enter "B" to show only the main log information. This parameter applies to all log archive utilities. Delete After Archive Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
BSDORESP	Restores data to the Domain log file. It uses the entire log data created by the BSDOARCP utility. It also generates a log record of the restore process. This utility has the following input field: Full Report Enter "F" to display the full details of all the data being logged. Enter "B" to show only the main log information. This parameter applies to all log archive utilities.

Spectrum Group Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description						
BSGRARCP	<p>Allows the Spectrum Group log data to be archived to a workfile and optionally deleted from the Spectrum Group log, based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p> <table><tr><td>Input End Date</td><td>Enter the last LOG date to be archived. This parameter applies to all log archive utilities.</td></tr><tr><td>Full Report</td><td>Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.</td></tr><tr><td>Delete After Archive</td><td>Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.</td></tr></table>	Input End Date	Enter the last LOG date to be archived. This parameter applies to all log archive utilities.	Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.	Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
Input End Date	Enter the last LOG date to be archived. This parameter applies to all log archive utilities.						
Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.						
Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.						
BSGRRESP	<p>Restores data to the Spectrum Group log file. It uses the entire log data created by the BSGRARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p> <table><tr><td>Full Report</td><td>Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.</td></tr></table>	Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.				
Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.						

Application Service Definition Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description				
BSIFARCP	<p>Allows the Application Service Definition log data to be archived to a workfile and optionally deleted from the Application Service Definition log based on a date. It also generates a log record of the archive process. It uses the following additional parameters</p> <p>This utility has the following input fields:</p> <table><tr><td>Report type</td><td>This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.</td></tr><tr><td>Delete After Archive</td><td>Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.</td></tr></table>	Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.	Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.				
Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.				
BSIFRESP	<p>Restores data to the Application Service Definition log file. It uses the entire log data created by the BSIFARCP utility. It also generates a log record of the restoration process.</p> <p>This utility has the following input field:</p> <table><tr><td>Report type</td><td>This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.</td></tr></table>	Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.		
Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.				

Spectrum Steplib Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description						
BSSDARCP	<p>Allows the Spectrum Steplib log data to be archived to a workfile and optionally deleted from the Spectrum Steplib log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p> <table><tr><td>Input End Date</td><td>Enter the last LOG date to be archived. This parameter applies to all log archive utilities.</td></tr><tr><td>Full Report</td><td>Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.</td></tr><tr><td>Delete After Archive</td><td>Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.</td></tr></table>	Input End Date	Enter the last LOG date to be archived. This parameter applies to all log archive utilities.	Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.	Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
Input End Date	Enter the last LOG date to be archived. This parameter applies to all log archive utilities.						
Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.						
Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.						
BSSDRESP	<p>Restores data to the Broker Steplib log file. It uses the entire log data created by the BSSDARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p> <table><tr><td>Full Report</td><td>Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.</td></tr></table>	Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.				
Full Report	Enter “F” to display the full details of all the data being logged. Enter “B” to show only the main log information. This parameter applies to all log archive utilities.						

User and Group Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description				
BSUSARCP	<p>Allows the User and Group log data to be archived to a workfile and optionally deleted from the User and Group log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p> <table><tr><td>Report type</td><td>This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.</td></tr><tr><td>Delete After Archive</td><td>Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.</td></tr></table>	Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.	Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.
Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.				
Delete After Archive	Mark this field if you want the log records to be deleted after they have been archived. This parameter applies to all log archive utilities.				
BSUSRESP	<p>Restores data to the User and Group log file. It uses the entire log data created by the BSUSARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p> <table><tr><td>Report type</td><td>This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.</td></tr></table>	Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.		
Report type	This field can be assigned the value “F” for full or “B” for brief. Enter “F” to include information related to the Interface and method data. Enter “B” to display only the log for the Application service header information.				

INDEX

Numerics

- 1:V fields
 - example of a PDA, 294
 - example of calling the Allocate method, 294
 - example of code to specify an array, 293
 - example of code to specify number of occurrences, 294
 - example of instantiating the PDA, 294
 - example of obtaining the bounds of an array, 295
 - example of passing the array, 295
 - example of using the same data area, 296
- 1:V overrides
 - panel, 137
- 1:V variables
 - Subprogram proxies
 - 1:V variable considerations, 136

A

- ABO Project
 - creating, 110
- ABO Wizard
 - using, 117
- ActiveX Business Object
 - role in architecture of Microsoft IIS, 41

- Adding
 - methods, 142
 - user exits, 134, 138
- Altered characters
 - description of, 227
- Application Administration Main menu
 - accessing, 221
- Application library
 - definition of, 321
- Application objects
 - troubleshooting, 232
- Application service
 - definition of, 321
- Application service definitions
 - accessing, 141
 - adding a method to, 142
 - invoking the Maintain Application Service Definitions panel, 142
 - list of methods, 141
- Application services
 - creating and selecting server components, 298
 - definition of, 321
 - external parameters, 299
 - global data areas, 299
 - parameter data areas, 299
 - subprogram behavior, 299
 - subprogram interface, 299
 - terminal I/O, 298
 - timing issues, 300
- Architecture
 - definition of, 321
 - diagram, 33

ASCII character set
translation of, 35–37

B

BDT_DATE, 171

BDTcontroller

See Business data type controller

Block handling

default methods, 146

overriding, 146

specifying on the server, 149

Browse command handler

definition of, 322

Browse data cache

definition of, 322

Browse dialog

definition of, 322

Browse dialogs

creating with individual models, 107

Browse process

definition of, 322

Browse subprogram proxy

number of occurrences, 136

specifying number of occurrences, 134

BSBLARCP

log utility, 366

BSBLRESP

log utility, 366

BSCTARCP

control record log utility, 367

BSCTRESP

control record log utility, 367

BSDOARCP

domain log utility, 368

BSDORESP

domain log utility, 368

BSGRARCP

group log utility, 369

BSGRRESP

group log utility, 369

BSIFARCP

Application service definition log
utility, 370

BSIFRESP

Application service definition log
utility, 370

BSSDARCP

steplib log utility, 371

BSSDRESP

steplib log utility, 371

BSUSARCP

user and group log utility, 372

BSUSRESP

user and group log utility, 372

Business data type

definition of, 322

Business data type (BDT)

definition of, 322

Business Data Type (BDT) controller

calling conversion routines, 159

convert from display, 160

convert to display, 160

converting in-place, 161

creating sample values, 162

error information properties, 164

ErrorCode, 164

ErrorLen, 164

ErrorMsg, 164

ErrorPos, 164

example of code, 165

syntax example, 158

using Natural formats, 163

Business Data Type (BDT) objects

BDTController, 176

BDTConversion, 176

- diagram of properties and methods, 177
- Business data type class
 - definition of, 322
 - definiton of, 322
- Business data type controller class
 - definition of, 322
- Business data type controller object
 - definition of, 323
- Business data type modifier
 - definition of, 323
- Business data type procedure
 - definition of, 323
- Business data types
 - benefits, 155
 - client framework, 158
 - client framework support, 156
 - composition of
 - conversion routine, 156
 - modifiers, 157
 - name, 156
 - creating
 - modifiers, 172
 - name, 172
 - Natural formats supported, 173
 - returning appropriate variant types, 173
 - customizing and creating, 172
 - overview, 154
 - registering, 181
 - example of code, 181
 - setting up in Predict, 48
 - supplied with CONSTRUCT
- Spectrum
 - Date, 170
 - Numeric, 169
 - Time, 168
- supplied with Construct Spectrum
 - Alpha, 167
 - Boolean, 167
 - Currency, 170

- used by client framework
 - diagram of, 158
 - using modifiers, 162
- Visual Basic data types, 155
 - diagram of, 156
- Business object
 - definition of, 323
 - setting up in Predict, 50
- Business-Object super model
 - before using, 90
 - application environment, 92
 - default values in Predict, 90
 - model defaults, 90
 - naming convention, 91
 - generating packages, 93
 - General Packages step, 96
 - New Package step, 98
 - Standard Parameters step, 94
 - overview of, 88
 - when to use, 89

C

- Call Interface, 344
- CallNat Method
 - syntax of, 277
- CALLNAT simulation, 39
- CallSystem Method
 - syntax of, 278
- Cardinality
 - definition of, 323
- Character translaton, 364
- Character UI
 - role in architecture of mainframe server, 34
- Checklists
 - application creation, 46

- Child model
 - definition of, 323
- Client applications
 - definition of, 324
 - developing
 - creating the user interface, 316
 - running applications, 318
 - writing code, 317
- Client calls
 - simulating for debugging, 219
- Client framework
 - definition of, 324
- Client/Server application
 - creating using Construct Spectrum tools, 18
 - creating without using Construct Spectrum's client framework, 19
- Code
 - preserving customizations to generated code, 71
 - protecting using implied user exits, 70
 - using the cst
 - PRESERVE tag, 71
- Command block
 - definition of, 324
- Command handlers
 - definition of, 324
- Command IDs
 - definition of, 324–325
- Communication errors
 - finding a list of communication errors, 204
 - handling, 203
 - list of possible origins, 203
 - list of severe errors, 203
 - retrieving information, 204
- Complex redefine
 - definition of, 325
- Compression
 - definition of, 325
- Compression/decompression of data, 35–37
- Configuration Editor
 - invoking, 60
- Construct Spectrum
 - definition of, 325
- Construct Spectrum Add-In
 - definition of, 325
- Construct Spectrum administration subsystem
 - definition of, 325
- Construct Spectrum for Visual Basic
 - documentation, 23
- Construct Spectrum SDK
 - documentation, 22
- Conventions
 - in this guide, 20, 22
- Conversation Factory, 363
- Conversion routines
 - creating, 178
 - handling errors from, 164
 - list of properties and methods, 178, 193
- Creating
 - applications, 245
 - assign values to fields in parameter data areas, 246
 - example of code to write to data areas, 246
 - check success of CALLNAT, 247
 - example of code to check CallNat, 247
 - create parameter data area instances, 245
 - example of code declaring variables, 245
 - example of creating a data area, 246
 - use the CallNat method on the client, 246

- example of code to use the CallNat method, 247
 - Construct Spectrum applications
 - without the client framework, 297
 - overview, 297
 - setting up server components, 298
 - domain for your application, 47
 - Customize
 - Customize Properties, 124
- D**
- Data compression/decompression, 35–37
 - Data encryption/de-encryption, 35
 - Data sizes tab
 - dialog, 225
 - Data translation, 35, 37
 - Data Type
 - description, 125
 - Database record
 - definition of, 325
 - DBID
 - definition of, 325
 - Debug data
 - Debug Library field, 215
 - generating, 208
 - writing to the source area
 - example of code in subprograms, 211
 - example of results from IF statement, 212
 - example of results without IF statement, 212
 - Debugging, 201
 - communication errors, 208
 - information, 37
 - on client and server, 208
 - list of error sources, 202
 - overview, 202
 - runtime errors, 208
 - traditional debugging tools, 204
 - where to find related information, 201
 - Debugging tools, 204
 - definition of, 326
 - INPUT statement, 217
 - on client and server
 - diagnostics program, 223
 - translations program, 227
 - simulating client calls, 219
 - traditional
 - DISPLAY statements, 204
 - INPUT statements, 204
 - Natural Debugging facility, 204
 - WRITE statements, 204
 - Defaulting from Predict
 - business object description, 50
 - GUI controls, 48
 - hold field, 49
 - primary key, 49
 - DEFINE PRINTER statement
 - using, 211
 - Defining
 - domain for your application, 54
 - security for your application, 57
 - steplib chain for your application, 52
 - Dependent object
 - definition of, 326
 - Deploying applications, 237
 - definition of, 326
 - distributing, 239
 - collecting files for installation, 239
 - creating the executable, 239
 - installing the client application, 240
 - running the application, 240
 - Descriptions
 - defaulting for business objects, 50
 - Developing environments
 - integrated tools for additional functionality, 27

- Development environments
 - Construct Spectrum Add-Ins in Visual Basic, 29
 - Construct Spectrum Administration subsystem, 28
 - Construct Spectrum options on the Add-Ins menu, 30
 - Construct Windows interface, 29
 - creating a web application, 31
 - using an HTML editor, 31
- Development process
 - steps involved in developing an application, 42
- Diagnostics program
 - diagram of data sizes tab, 225
 - diagram of initialize data tab, 226
 - diagram of subprogram proxy tab, 224
 - Dispatcher.CallNat, 223
 - Dispatcher.RequestProperty, 223
 - initialize data tab, 226
 - purpose of, 223
 - summary of diagnostic information, 223
 - using, 224
- Dialogs
 - definition of, 326
- Dispatch service data
 - role in architecture of mainframe server, 36
- Dispatcher
 - definition of, 326
- Dispatcher objects
 - troubleshooting, 233
- DISPLAY statement, 298
- Distributing applications, 239
 - collecting files for installation, 239
 - creating the executable, 239
 - installing the client application, 240
 - running applications, 240
- Documentation
 - Construct Spectrum, 23

- Construct Spectrum Software Development Kit, 22
- Natural Construct, 23
 - related, 22
- Domains
 - defining, 54
 - defining security for, 57
 - definition of, 326
- Double byte character sets DBCS
 - definition of, 326
- Downloading data
 - definition of, 326

E

- EBCDIC character set
 - translation of, 35–37
- Enabling trace options in subprogram proxy, 136
- Encapsulation
 - definition of, 327
- Encrypt and decrypt data, 37
- Encrypting data
 - definition of, 327
- Encryption/de-encryption of data, 35
- Entire Broker
 - encapsulation of calls, 39
 - role in architecture of mainframe server, 36
 - role in architecture of Microsoft IIS, 40
 - role in architecture on Windows platform, 38
 - SPUETB subprogram, 350
- Entire Broker service settings
 - definition of, 327
- Entire Broker stub
 - definition of, 327

Error categories, 206
 communication, 206
 runtime, 206
 Spectrum system messages, 206

Error handling, 37

ErrorCode, 164

Errors.bas
 description, 116

ETBCB Data Area, 358

Event
 definition of, 327

Externally-creatable object
 definition of, 327

F

Field
 definition of, 327

Field headings
 setting up in Predict, 47

FieldRef property
 diagram of creating two data
 areas, 292
 diagram of fields defined as objects to
 CallNat, 289
 diagram of fields passed to
 CallNat, 288
 diagram of using, 291
 example of code, 291
 example of code to pass individual
 fields, 287
 example of using the CUSTA Natural
 data area, 292
 example of Visual Basic Code, 293
 syntax for, 290

File volume information
 specifying in Predict, 51

FNR
 definition of, 327

Foreign fields key field
 definition of, 327

Form
 definition of, 328

G

Generating
 a subprogram proxy, 132
 debug data, 208
 saving parameter and debug
 data, 208
 definition of, 328
 Generate Trace Code field, 213
 Generated module
 definition of, 328
 package modules
 in the Construct Windows
 interface, 102
 in the Generation subsystem, 104
 subprogram proxy
 in the Construct Windows
 interface, 134

Get/Let
 description, 125

Global data areas, 299

Globals.bas
 description, 116

Grids
 control, 329
 definition of, 329

Group, user
 definition of, 329

Grouping related business objects, 54

GUI
 control override
 definition of, 329

definition of, 329

GUI controls

setting up in Predict, 48

GUI dialog

role in architecture on Windows

platform, 39

H

Hold field default

setting up in Predict, 49

Hook

definition of, 329

Host

definition of, 329

I

Initialize data tab

purpose of, 226

INPUT statement, 298

as debugging tool, 217

Installing

See Distributing applications, 239

Instantiation

definition of, 330

Internationalizing

definition of, 330

Internet/Intranet

support, 41

Invoke Proxy function

accessing, 220

panel, 221

J

Job control language (JCL)

definition of, 330

K

Keys

defaulting primary, 49

Keywords

definition of, 330

L

Level 1 Block Optimization, 269

diagram of client and server as sender
and receiver, 271

directional attributes

example of code showing directional
attribute, 270

list of directional attributes, 269

Level 1 data block

definition of, 330

optimization

definition of, 330

Libraries

adding to your steplib chain, 54

Library Image File

definition of, 330

Library image files

simulated PDAs in, 38

LIFDefinitions.bas

description, 116

Localization

definition of, 330

Log utilities, 366

Lookups
definition of, 331

M

Maintain Domains Table panel, 56

Maintain Steplib Table
panel, 54

Maintain User Table
panel
accessing, 214

Maintenance dialog
definition of, 331

Maintenance subprogram proxy
level 1 blocks sent for default
methods, 146

Mapper function, 163

MDI child
definition of, 331

MDI frame
definition of, 331

MDI parent
definition of, 331

Menu
definition of, 331

Menu bar
definition of, 332

Message handling, 37

Message Protocol
SYSERR-LIBRARY(A8), 343

Message protocol
MESSAGE(A1/1
V), 343
MSG-NR(N4), 343
REPLICA-ID(A32), 343
RESPONSE-CODE(N4), 343
SIGNATURE(A6), 343

SPECTRUM-SERVICE(A32), 343

Metadata
definition of, 332

Methods
abort, 283
adding, 142
creating, 143
updating application service
definitions, 143
updating LIFs, 144
allocate, 254
commit, 283
definition of, 332
GetField, 257
initialize, 254
Reset, 258
SetField, 259
StartTransaction, 283

Microsoft Visual Basic, 297

Middleware
how it works with Construct
Spectrum, 26

Models
definition of, 332

Modules
definition of, 332

Multi-level security
definition of, 332

Multiple-document interface
definition of, 332

Multi-tasking verification utility, 365

N

Natural Construct
documentation, 23
nucleus
definition of, 333

Natural Construct Administration and Modeling User's Manual, 23

Natural Construct Help Text User's Manual, 23

NATURAL data area simulation

 NaturalDataArea object

 example of code for redefining, 261

 other features, 259

Natural data area simulation, 250

 application object

 example of code to declare and

 initialize the application object, 254

 application object properties or methods, 254

 allocate method, 254

 initialize method, 254

 LIFDirectory property, 254

 MainLibrary property, 254

 creating NaturalDataArea

 objects, 255

 syntax for allocate method, 255

 data area definitions, 250

 example of code for data area

 definition, 251

 list of features, 251

 data area simulation objects, 252

 diagram of objects in data area

 simulation, 253

 definition of, 250

 diagram of components, 250

 NaturalDataArea class

 list of properties or methods, 256

 NaturalDataArea object

 class, 255

 example of a data area

 definition, 263

 example of code showing side effects of Redefining, 261

 example of code using structure

 name as a qualifier, 260

 example of reading arrays with the GetField method, 262

 example of reading occurrences of

 the Item array, 263

 example of specifying a field with occurrences, 262

 NaturalFieldDef class, 264

 list of properties, 265

 Natural Debugging facility

 definition of, 333

 Natural EDIT command, 216

 Natural LIST command, 216

 Natural security

 user information, 215

 Natural source areas, 298

 Natural subprogram

 role in architecture of mainframe server, 34

 Natural subprograms

 example of creating, 300

 example PDA, 301

 panel, 302

 NaturalDataArea object

 troubleshooting, 232

 Node

 definition of, 333

 Nodes

 marking for refresh, 84

 removing from cache, 85

 Nucleus

 definition of, 333

O

Object

 definition of, 333

Object factory

 definition of, 333

Object library

 definition of, 334

OLE automation server, 297
definition of, 334

Overflow conditions
definition of, 334

Overriding
1:V variables, 136
domain steplib chain, 144

P

Packages
definition of, 334

Parameter alignment problems
diagnosing, 223

Parameter and debug data
accessing the Maintain User Table
panel, 214
using, 213

Parameter data areas
data size limitations, 299
panel, 301
simulation of by Spectrum dispatch
client, 38

Parameters
externalizing, 299

Parent model
definition of, 334

Partner products
what Construct Spectrum works
with, 26

Ping
definition of, 334

Platforms
definition of, 334

Predict data dictionary
how it works with Construct
Spectrum, 26

Predict set up tasks
business data types, 48
default business object description, 50
default GUI controls, 48
default hold field, 49
default primary key, 49
field headings, 47
file volume information, 51
verification rules, 49, 51

Preface, 15

Prerequisites, 16

Preserving
characters
definition of, 227

Primary keys
defaulting from Predict, 49

PRINT statement, 298

Printable characters
description of, 227

Programming languages
incorporating with Construct
Spectrum, 27

Projects
definition of, 334
window
definition of, 335

Properties
CheckFieldSpec, 256, 264
Decimals, 265
DefinedRank, 265
definition, 256, 264
definition of, 335
field, 256
FieldDef, 257, 264
FieldDefs, 257, 264
FieldRef, 257
Format, 265
FormatLength, 265
FromIndex and ThruIndex, 266
Length, 266
Level, 266

- LevelTypeTrail, 266
- LibraryImageFile, 258
- LIFDirectory, 254
- MainLibrary, 254
- Name, 258, 267
- PackedData, 258
- PackedDataLength, 258, 264
- Rank, 267
- Redefined, 268
- Structure, 268
- ThruIndex, 268
- TransactionActive, 283
- window
 - definition of, 335
- Property Name
 - description, 125

R

- Remote call
 - definition of, 335
- Reports
 - using, 77
- RequestProperty properties, 233
- Resource
 - definition of, 335
- Running
 - applications, 318
 - definition of, 335
- Runtime errors
 - results, 219
 - where to find a list of, 206–207

S

- SDC
 - See* Spectrum Dispatch Client

- Security
 - defining for a domain, 57
 - security cache
 - definition of, 335
- Security services
 - role in architecture, 36
- Server
 - application
 - definition of, 335
 - definition of, 335
 - setup, 298
- Service
 - definition of, 336
 - exit
 - definition of, 336
 - log
 - definition of, 336
- Set, 227
- Set up checklists
 - see* Checklists, 46
- Setting
 - trace options, 208
- Setting up
 - security for your application, 52
- Shutdown
 - definition of, 336
- SPAETB Data Area, 352
- Specifying
 - block handling on the server, 149
 - General Package Parameters
 - Business-Object super model wizard, 95
 - New Package Parameters
 - Business-Object super model wizard, 101
 - overrides, 148
 - Specific Package Parameters
 - Business-Object super model wizard, 98

- Standard Package parameters
 - Business-Object super model wizard, 93
- Specifying defaults
 - hold key, 50
 - primary key, 49
- Spectrum administration
 - role in architecture of mainframe server, 36
- Spectrum control record
 - definition of, 336
- Spectrum Dispatch Client, 243
 - advanced features, 287
 - 1 to V fields, 293
 - FieldRef property, 287
 - application service definitions, 271
 - example in a library image file, 273
 - list of information needed in, 271–272
 - client/server communication, 268
 - application service, 249
 - components, 249
 - data area simulation, 250
 - level 1 block optimization, 269
 - client/server communication components
 - definitions, 249
 - dispatch service definitions, 249
 - dispatcher objects, 249
 - creating applications
 - See* creating applications, 245
 - data area simulation components, 249
 - data area allocator, 249
 - data area definitions, 249
 - data area objects, 249
 - definition of, 336
 - Dispatcher objects and dispatch service definitions, 274
 - compression and encryption, 282
 - database transaction control, 283
 - diagram of dispatcher objects, 275
 - diagram of timeout functionality, 280
 - example of code to create dispatcher objects, 276
 - example of implementing level 1 block optimization, 277
 - example of resuming a call, 281
 - list of error types, 284
 - remote subprogram invocation, 277
 - service selection, 276
 - timeout, retry, and resume handling, 278
 - tracing, 282
 - user identification and authentication application properties, 285
 - functions of
 - client/server communication, 249
 - Natural data area simulation, 249
 - initializing
 - Options dialog, 316
 - library image files and the steplib chain, 286
 - syntax of the steplib definition, 286
 - overview, 244
 - example of code calling subprogram, 244
 - example of code with defined parameter data, 244
 - properties, 231
 - role in architecture of Microsoft IIS, 40
- Spectrum dispatch client
 - CALLNAT simulation, 39
 - database transaction control, 39
 - encapsulation of Entire Broker calls, 39
- Spectrum Dispatch Client (SDC)
 - role in architecture on Windows platform, 38
- Spectrum Dispatch Service
 - definition of, 336

- running online, 217
- Spectrum dispatch service
 - role in architecture of mainframe server, 35
- Spectrum security service
 - definition of, 336
- Spectrum security services
 - role in architecture of mainframe server, 36
- Spectrum service settings
 - definition of, 336
- Spectrum XML Cache Viewer
 - overview of, 83
 - refreshing, 84
- SPSTLATE utility, 364
- SPUASCII
 - determining a character set, 364
- SPUETB
 - features and benefits, 350
 - using, 359
- SPUETB Interface, 351
- SPUETB utility, 350
- SPUREPLY Subprogram, 342
- SPUREPLY utility, 342
- Statements
 - DISPLAY, 298
 - INPUT, 298
 - PRINT, 298
 - STACK TOP DATA, 298
 - WRITE, 298
- Status bar
 - definition of, 337
- Steplib chain
 - defining for your application, 52
 - definition of, 337
- Sub Main procedure
 - definition of, 337
- Subprogram proxies
 - and Spectrum dispatch service, 35
 - generating using models, 303
 - invoking online, 219
 - setting trace code option, 136
- Subprogram proxy
 - adding user exits to, 138
 - definition of, 337
 - generating from the Construct Windows interface, 139
 - methods generated by, 141
 - overview
 - application service definition
 - methods
 - See Application service definitions, 141
 - prerequisites
 - generating subprograms and object PDAs, 141
 - Spectrum administration subsystem data files, 141
 - role in architecture of mainframe server, 34
 - tab, 224
 - versioning, 151
 - security implications, 151
- Subprogram proxy model
 - generating
 - considerations for, 132
 - standard parameters, 134
 - standard parameters step, 134
 - generating with, 132
 - overview of, 130
- Subprograms
 - behavior, 299
 - interfacing, 299
- Super model
 - definition of, 337
- Supported methods
 - LOOKUP-MESSAGE, 342
 - SEND-MESSAGE-ONLY, 343
 - SEND-MESSAGE-ONLY-WITHOUT-EOC, 343

SEND-REPLY, 342
SEND-WITHOUT-EOC, 342

T

Target module
 definition of, 337
Target object
 definition of, 337
Target subprogram
 definition of, 338
Terminal I/O, 298
Timing issues, 300
Toolbar
 buttons
 definition of, 338
 definition of, 338
Toolkit
 definition of, 338
Trace options
 definition of, 338
 dialog, 209
 setting, 208
Trace-Option(1), 209
 purpose of, 209
Trace-Option(2), 213
 and Generate Trace Code field, 213
 possible values for, 213
 purpose of, 213
Translation mappings
 dialog, 228
Translation of character sets, 35–37
Translations program, 227
 character sets, 227
 altered, 227
 preserved, 227
 printable, 227
 purpose of, 227

translation tables, 227
 function of, 227

Troubleshooting
 Construct Spectrum Add-In, 231
 Construct Spectrum dispatch client
 properties, 231

Type library
 definition of, 338

Types of applications
 client/server applications, 32
 web applications, 32

U

Updating
 application service definitions, 143
 LIF files, 144

Uploading
 definition of, 338

User interface
 creating
 example form, 317
 creating the, 316

Using BDTs
 handling runtime errors, 174
 one conversion routine — multiple
 BDTs, 184
 overriding, 185
 placing the conversion routine, 184
 referencing
 in Predict, 187
 in your application, 186
 retrieving error information, 185

Utilities
 Conversation Factory, 363
 log, 366
 multi-tasking verification, 365
 SPSTLATE subprogram, 364
 SPUETB subprogram, 350

SPUREPLY subprogram, 342

Utility.bas
description, 116

V

Variants

definition of, 338

VB-Client-Server-Super-Model

definition of, 339

Verification rules

definition of, 339

setting up in Predict, 49, 51

Visual Basic browse object

definition of, 339

Visual Basic business object

definition of, 339

role in architecture of Windows
platform, 39

Visual Basic maintenance

business object

definition of, 339

Volume information

see File volume information

W

Web application

role in architecture of Microsoft
IIS, 41

Wildcards

definition of, 340

WRITE statement, 298

Writing code, 317